Deep Dive - Serverless Websockets in AWS

CHS AWS Meetup - 12/10/2019

We’re hiring! Hurray!

Snag Careers!

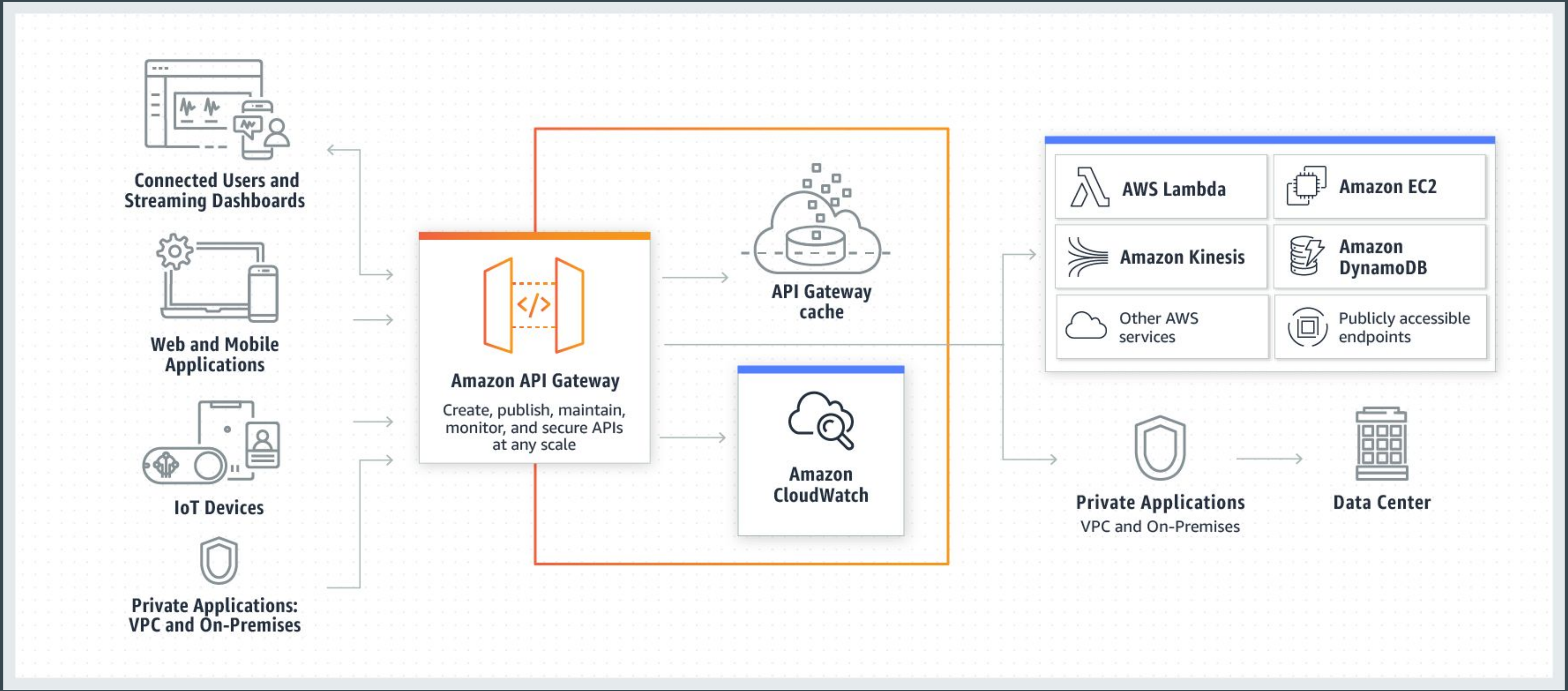
To be covered:

- WebSockets with AWS API Gateway
- Serverless Framework
- Building a serverless WebSocket server
- Building a WebSocket producer
    - Python
- Building a WebSocket consumer
    - Python
    - TypeScript

Not covered:

- WebSockets with AWS ALBs, Containers, Ec2, etc.

AWS API Gateway

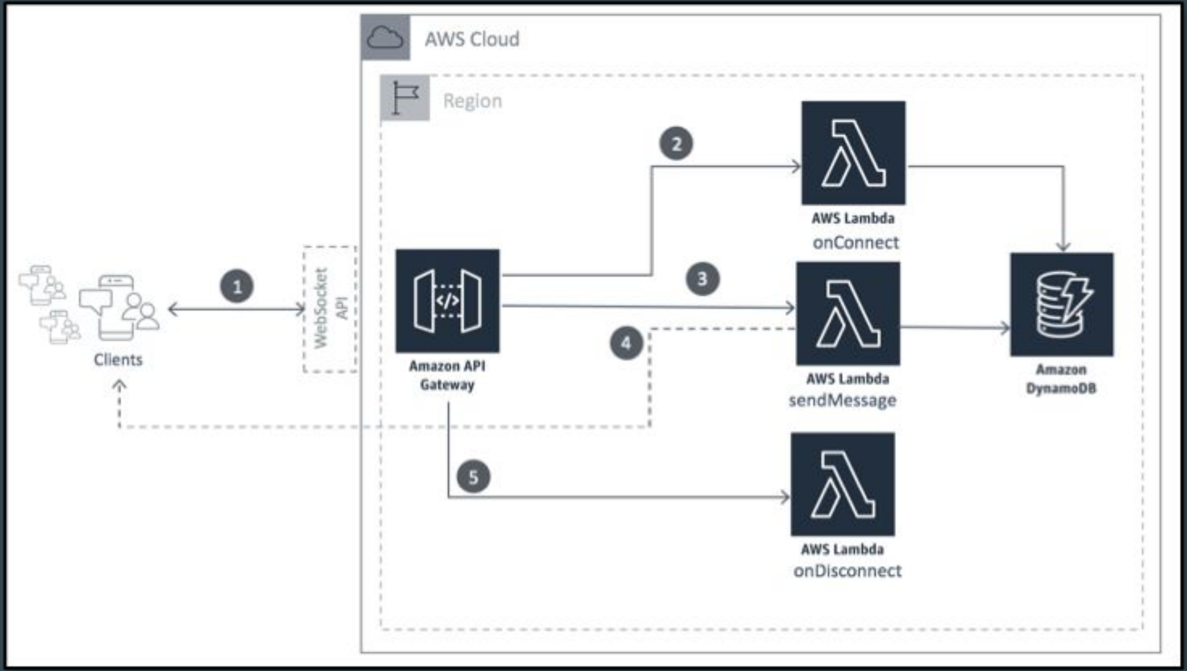

Connected Users and Streaming Dashboards
Web and Mobile Applications
IoT Devices
Private Applications: VPC and On-Premises
Amazon API Gateway
Create, publish, maintain, monitor, and secure APIs at any scale
API Gateway cache
Amazon CloudWatch
AWS Lambda
Amazon EC2
Amazon Kinesis
Amazon DynamoDB
Other AWS services
Publicly accessible endpoints
Private Applications
VPC and On-Premises
Data Center

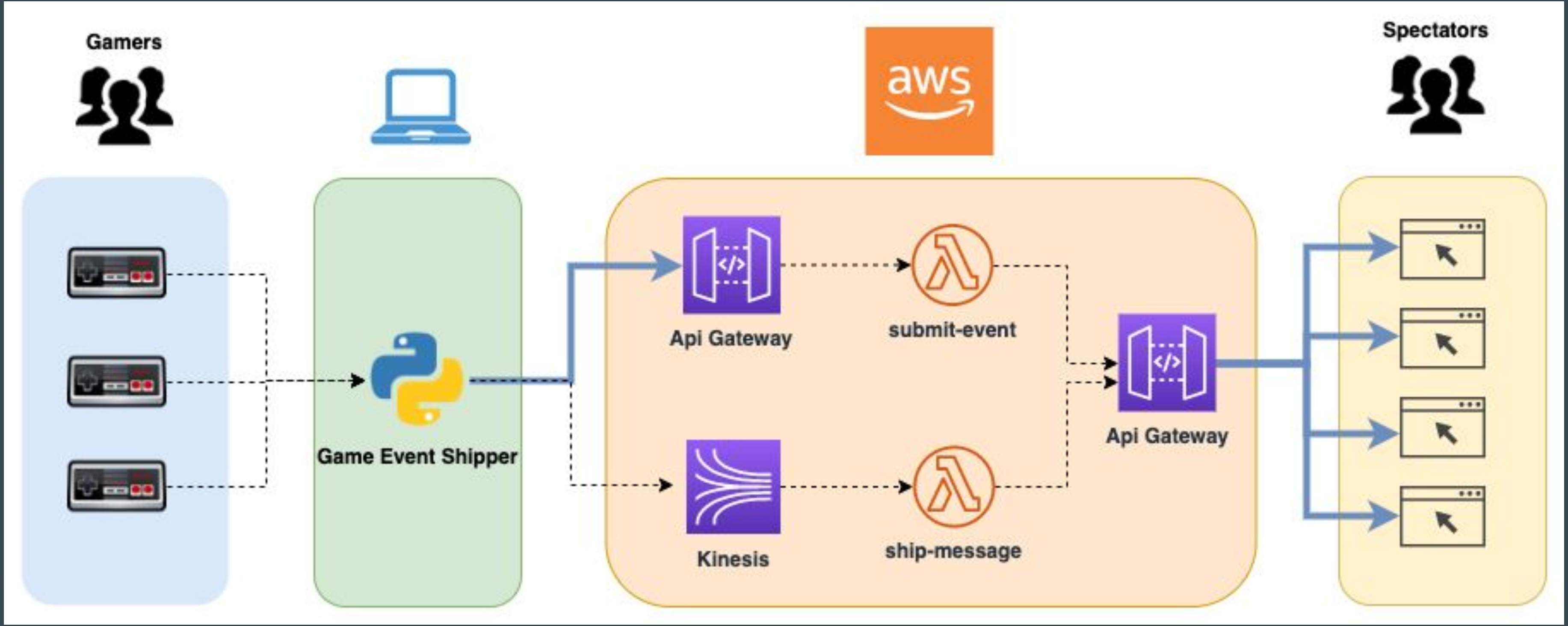
API Gateway Websockets

- Manages WebSocket connections on your behalf
- Exposes an API to interact with open connections

Pricing:

- $1 per million messages.
- $0.25 per million connection minutes

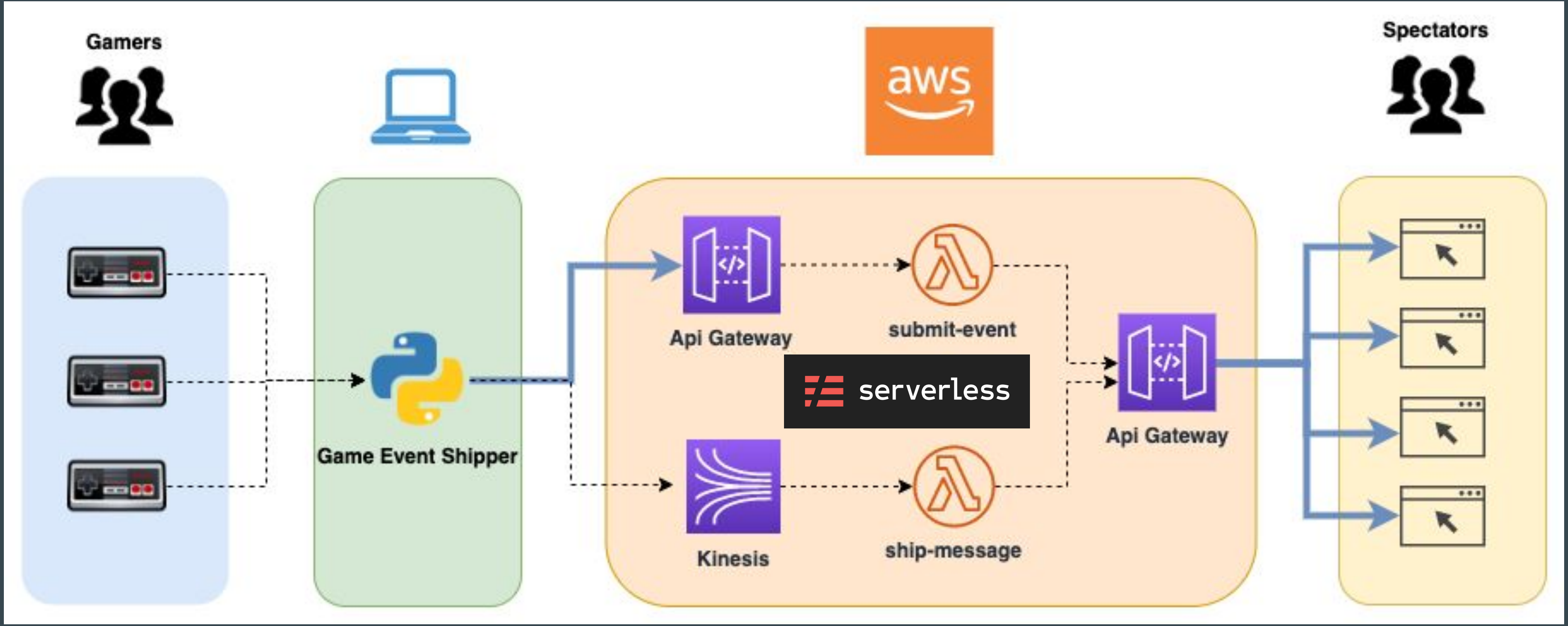




Snagajob Tumble Race

Serverless Framework

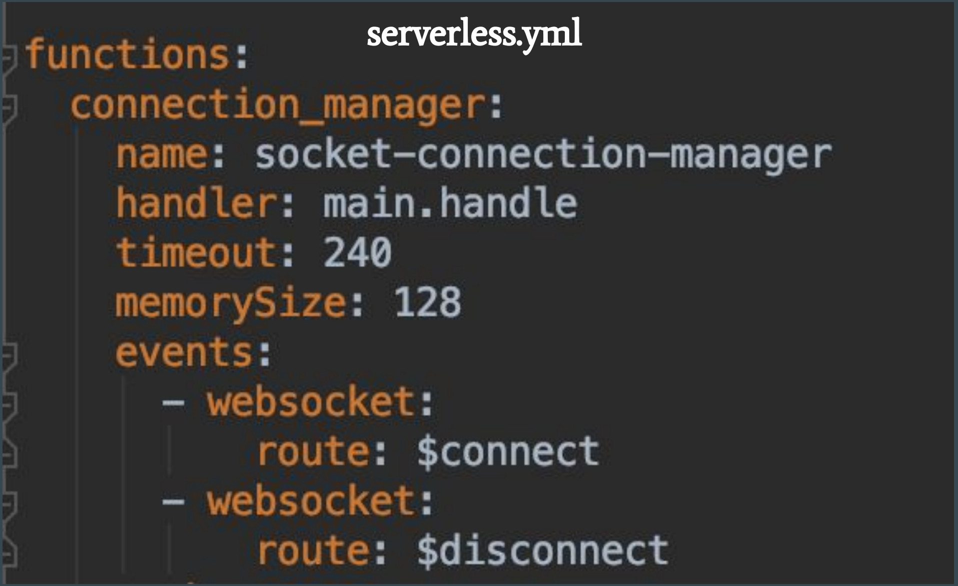
- Helps you build serverless applications in AWS, Azure, and more.
- Manage your code & infrastructure in one place
- Built-in CLI provides support for deployment across different environments (stages)
- Open source & community supported.
- Open source plugin library
- Supports AWS API Gateway w/ Websockets



Serverless provisions all the AWS stuff.

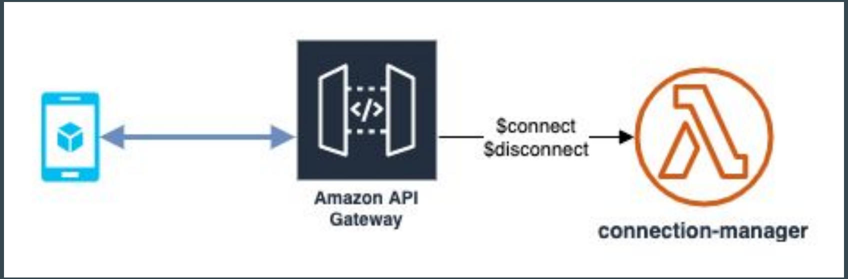
Building our "serverless server"

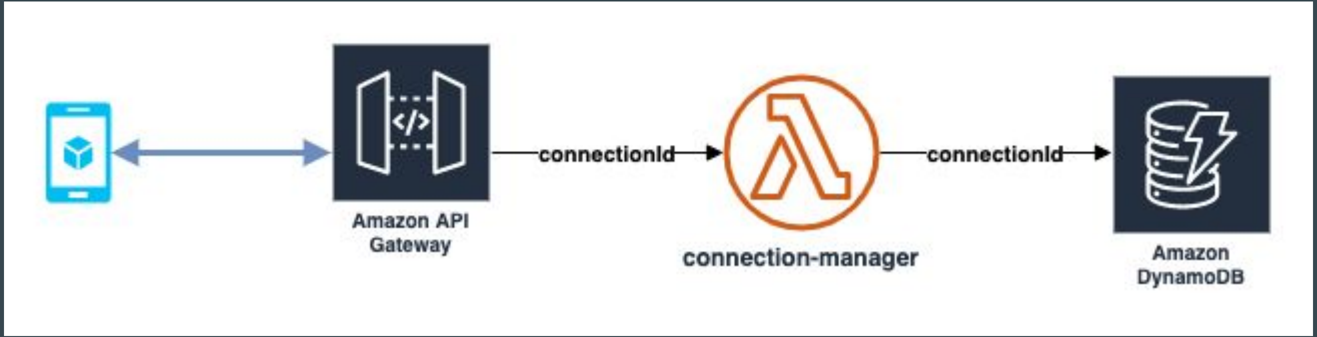
- Implement **$connect** and **$disconnect** routes
- Perform authentication
- Persist connectionId to a database



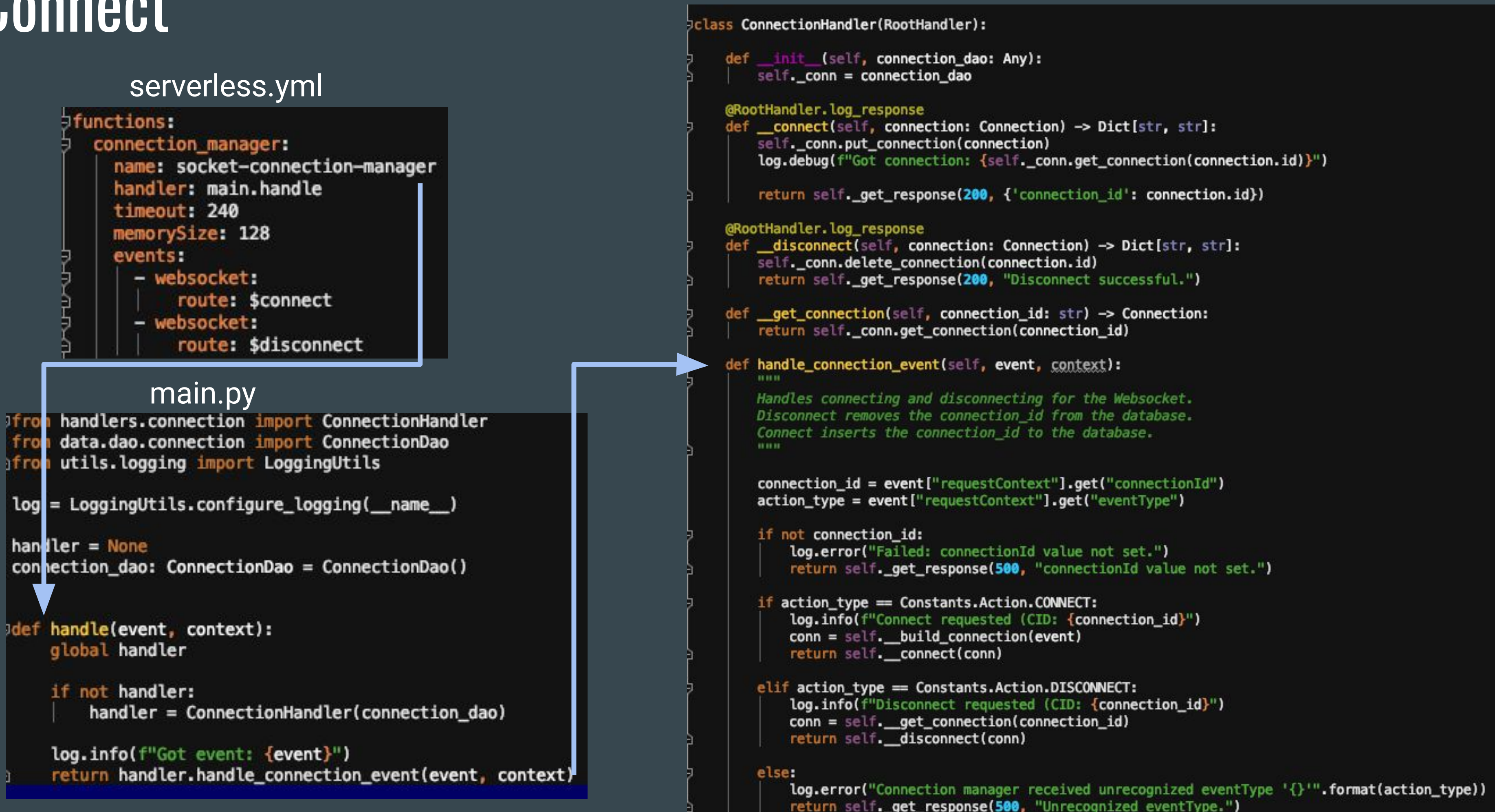
serverless.yml

```yaml
functions:
  connection_manager:
    name: socket-connection-manager
    handler: main.handle
    timeout: 240
    memorySize: 128
    events:
      - websocket:
          route: $connect
      - websocket:
          route: $disconnect
```





Connect



serverless.yml

```yaml
functions:
  connection_manager:
    name: socket-connection-manager
    handler: main.handle
    timeout: 240
    memorySize: 128
    events:
      - websocket:
          route: $connect
      - websocket:
          route: $disconnect
```

main.py

```python
from handlers.connection import ConnectionHandler
from data.dao.connection import ConnectionDao
from utils.logging import LoggingUtils

log = LoggingUtils.configure_logging(__name__)

handler = None
connection_dao: ConnectionDao = ConnectionDao()


def handle(event, context):
    global handler

    if not handler:
        handler = ConnectionHandler(connection_dao)

    log.info(f"Got event: {event}")
    return handler.handle_connection_event(event, context)
```

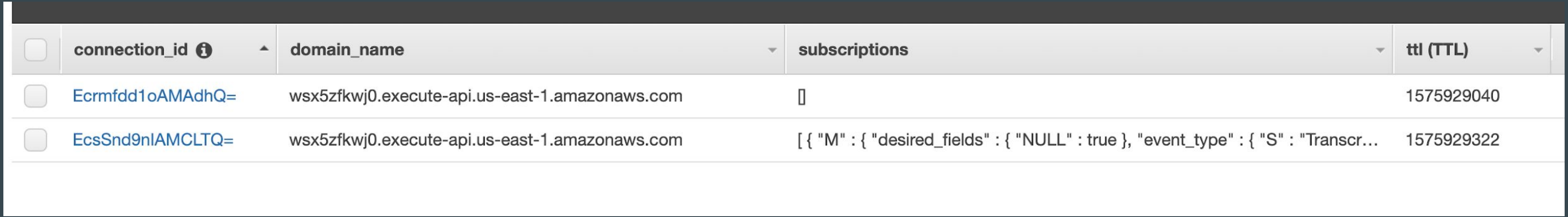
connection_handler.py

```python
class ConnectionHandler(RootHandler):

    def __init__(self, connection_dao: Any):
        self._conn = connection_dao

    @RootHandler.log_response
    def __connect(self, connection: Connection) -> Dict[str, str]:
        self._conn.put_connection(connection)
        log.debug(f"Got connection: {self._conn.get_connection(connection.id)}")

        return self._get_response(200, {'connection_id': connection.id})

    @RootHandler.log_response
    def __disconnect(self, connection: Connection) -> Dict[str, str]:
        self._conn.delete_connection(connection.id)
        return self._get_response(200, "Disconnect successful.")

    def __get_connection(self, connection_id: str) -> Connection:
        return self._conn.get_connection(connection_id)

    def handle_connection_event(self, event, context):
        """
        Handles connecting and disconnecting for the Websocket.
        Disconnect removes the connection_id from the database.
        Connect inserts the connection_id to the database.
        """

        connection_id = event["requestContext"].get("connectionId")
        action_type = event["requestContext"].get("eventType")

        if not connection_id:
            log.error("Failed: connectionId value not set.")
            return self._get_response(500, "connectionId value not set.")

        if action_type == Constants.Action.CONNECT:
            log.info(f"Connect requested (CID: {connection_id}")
            conn = self.__build_connection(event)
            return self.__connect(conn)

        elif action_type == Constants.Action.DISCONNECT:
            log.info(f"Disconnect requested (CID: {connection_id}")
            conn = self.__get_connection(connection_id)
            return self.__disconnect(conn)

        else:
            log.error("Connection manager received unrecognized eventType '{}'".format(action_type))
            return self._get_response(500, "Unrecognized eventType.")
```

Connections Table

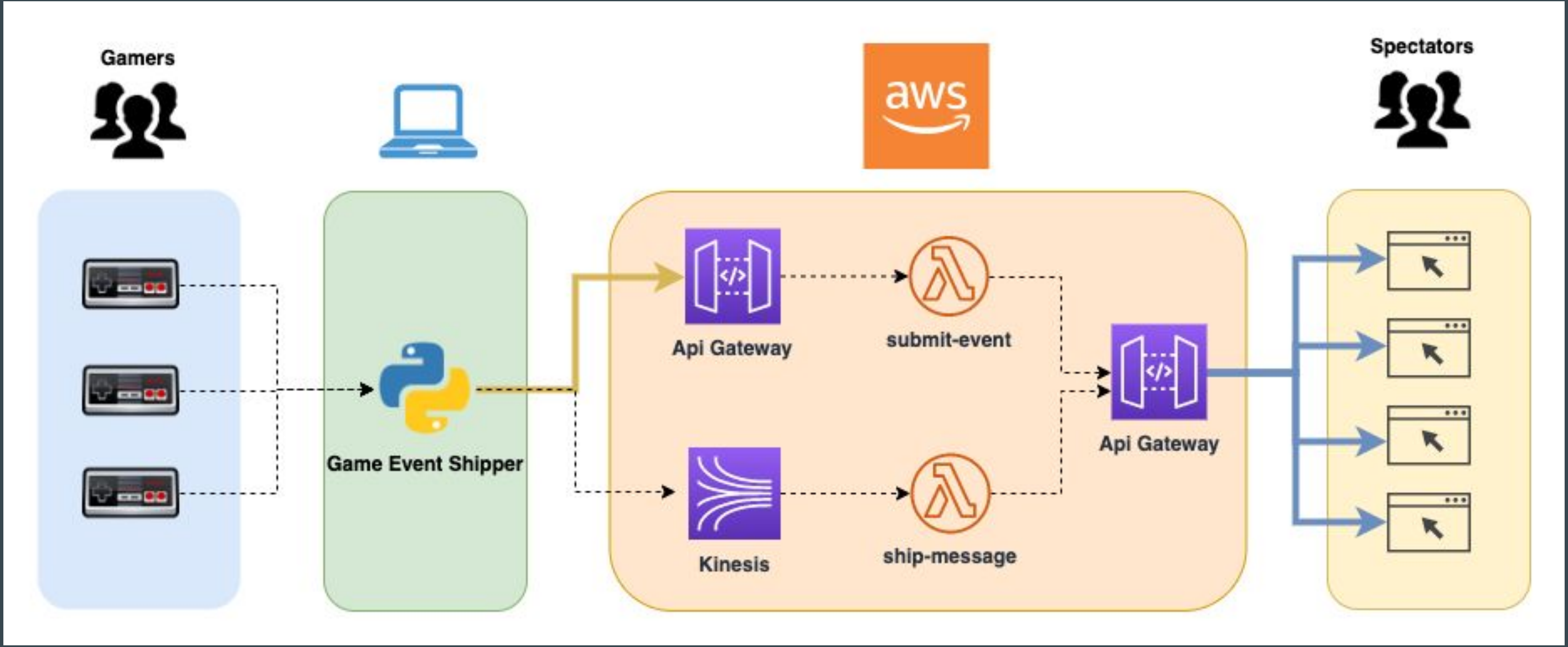
Maintains a list of all open WebSocket connections.

Includes both game VIEWERS and game event SHIPPERS



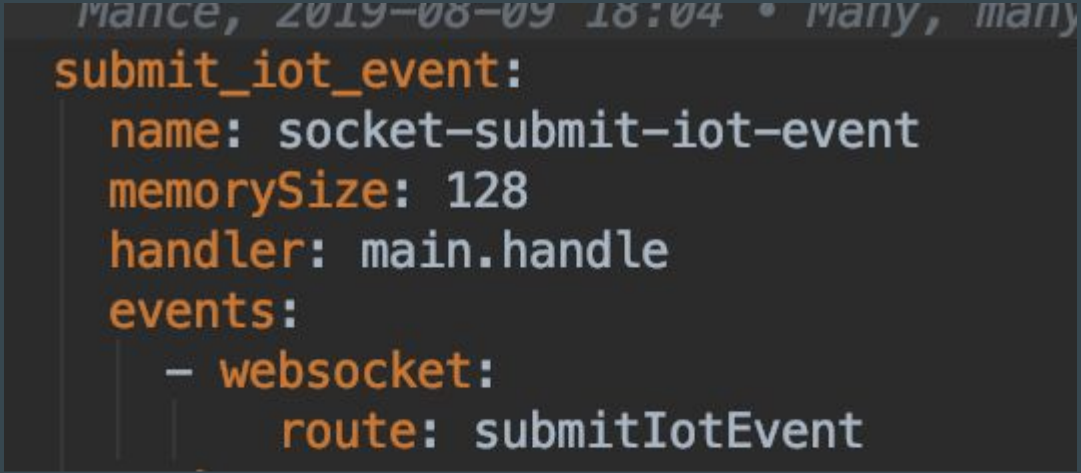
connection_id	domain_name	subscriptions	ttl (TTL)
Ecrmfdd1oAMAdhQ=	wsx5zfkwj0.execute-api.us-east-1.amazonaws.com	[]	1575929040
EcsSnd9nIAMCLTQ=	wsx5zfkwj0.execute-api.us-east-1.amazonaws.com	[{ "M" : { "desired_fields" : { "NULL" : true }, "event_type" : { "S" : "Transcr...	1575929322

We’re connected, now we need to receive events.



Adding a custom route so clients can submit events.

- Define a new new function + handler in your serverless.yml

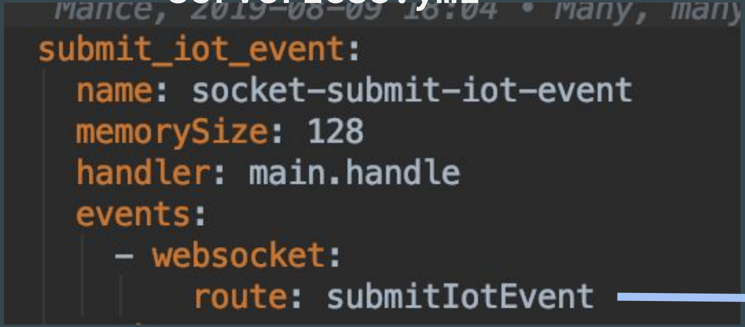


```
Mance, 2019-08-09 18:04 • Many, many
submit_iot_event:
  name: socket-submit-iot-event
  memorySize: 128
  handler: main.handle
  events:
    - websocket:
        route: submitIotEvent
```

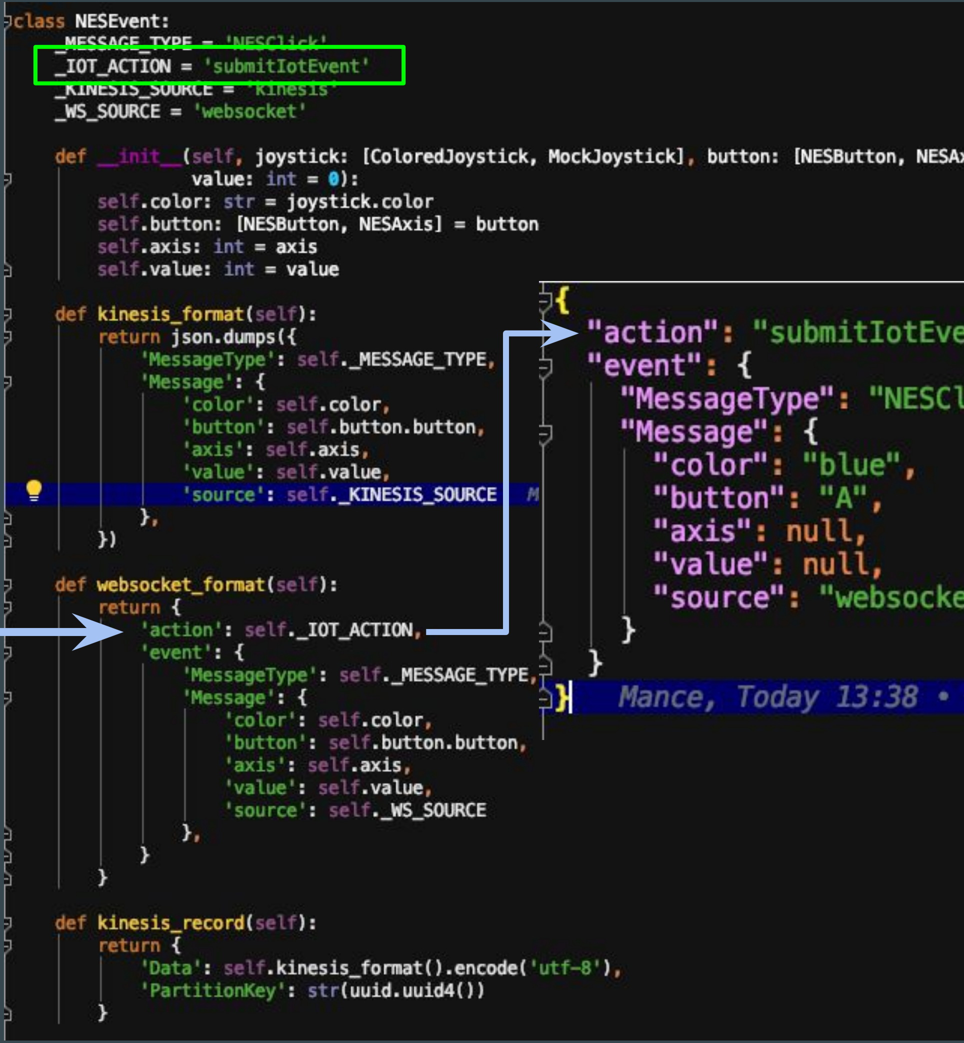
Calling a custom route

- Send JSON over the open websocket
- JSON must have a top level property of **‘action’** which defines the intended route.

serverless.yml



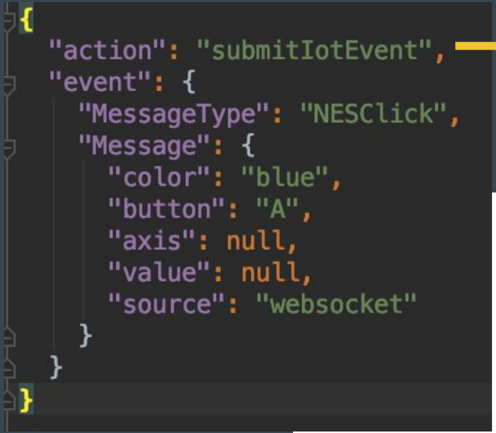
```
Mance, 2019-08-09 18:04 • Many, many
submit_iot_event:
  name: socket-submit-iot-event
  memorySize: 128
  handler: main.handle
  events:
    - websocket:
        route: submitIotEvent
```



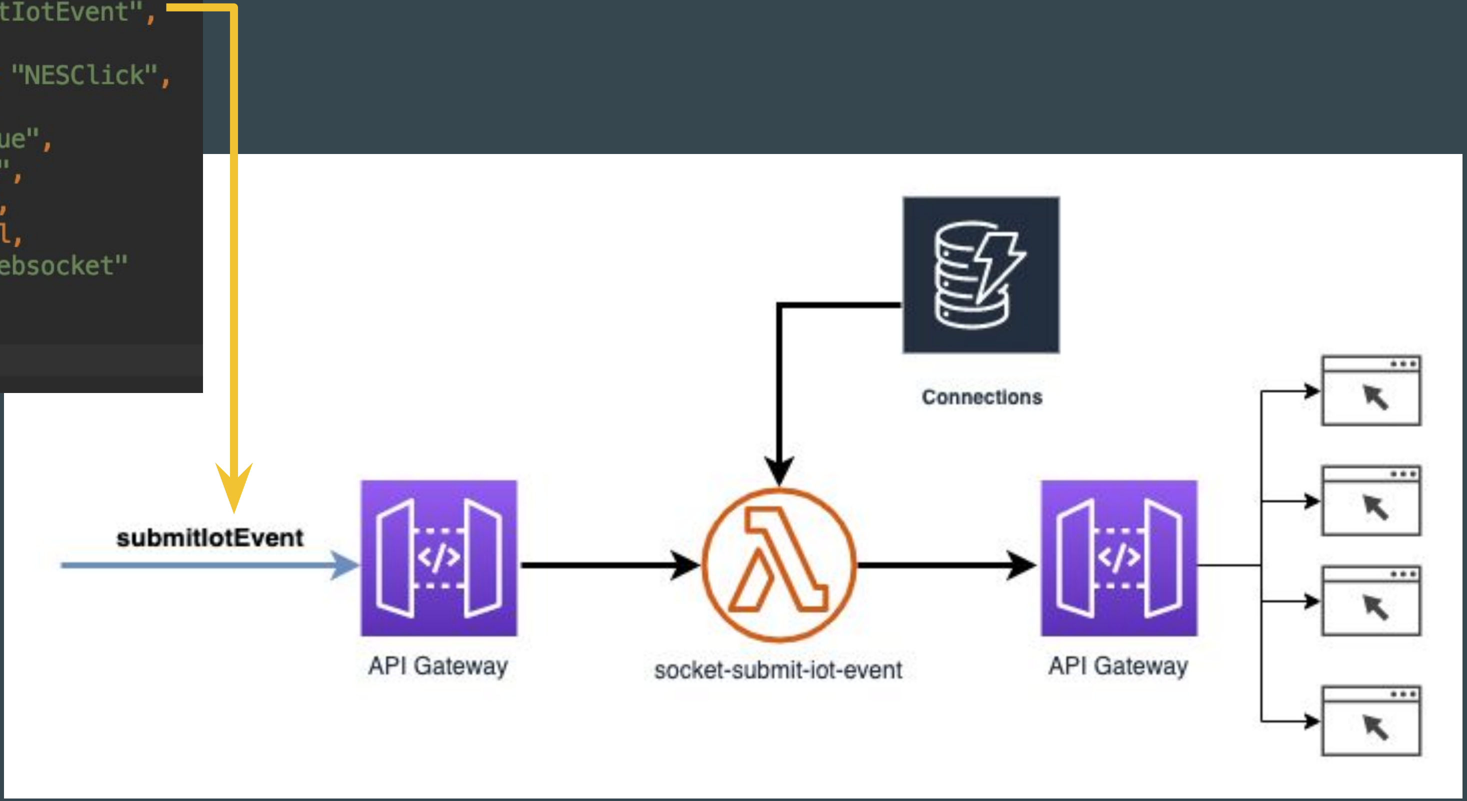
```
class NESEvent:
    _MESSAGE_TYPE = 'NESClick'
    _IOT_ACTION = 'submitIotEvent'
    _KINESIS_SOURCE = 'kinesis'
    _WS_SOURCE = 'websocket'

    def __init__(self, joystick: [ColoredJoystick, MockJoystick], button: [NESButton, NESAx
                 value: int = 0):
        self.color: str = joystick.color
        self.button: [NESButton, NESAxis] = button
        self.axis: int = axis
        self.value: int = value

    def kinesis_format(self):
        return json.dumps({
            'MessageType': self._MESSAGE_TYPE,
            'Message': {
                'color': self.color,
                'button': self.button.button,
                'axis': self.axis,
                'value': self.value,
                'source': self._KINESIS_SOURCE
            },
        })

    def websocket_format(self):
        return {
            'action': self._IOT_ACTION,
            'event': {
                'MessageType': self._MESSAGE_TYPE,
                'Message': {
                    'color': self.color,
                    'button': self.button.button,
                    'axis': self.axis,
                    'value': self.value,
                    'source': self._WS_SOURCE
                },
            }
        }

    def kinesis_record(self):
        return {
            'Data': self.kinesis_format().encode('utf-8'),
            'PartitionKey': str(uuid.uuid4())
        }
```

```
{
  "action": "submitIotEve
  "event": {
    "MessageType": "NESCl
    "Message": {
      "color": "blue",
      "button": "A",
      "axis": null,
      "value": null,
      "source": "websocke
    }
  }
}    Mance, Today 13:38 •
```

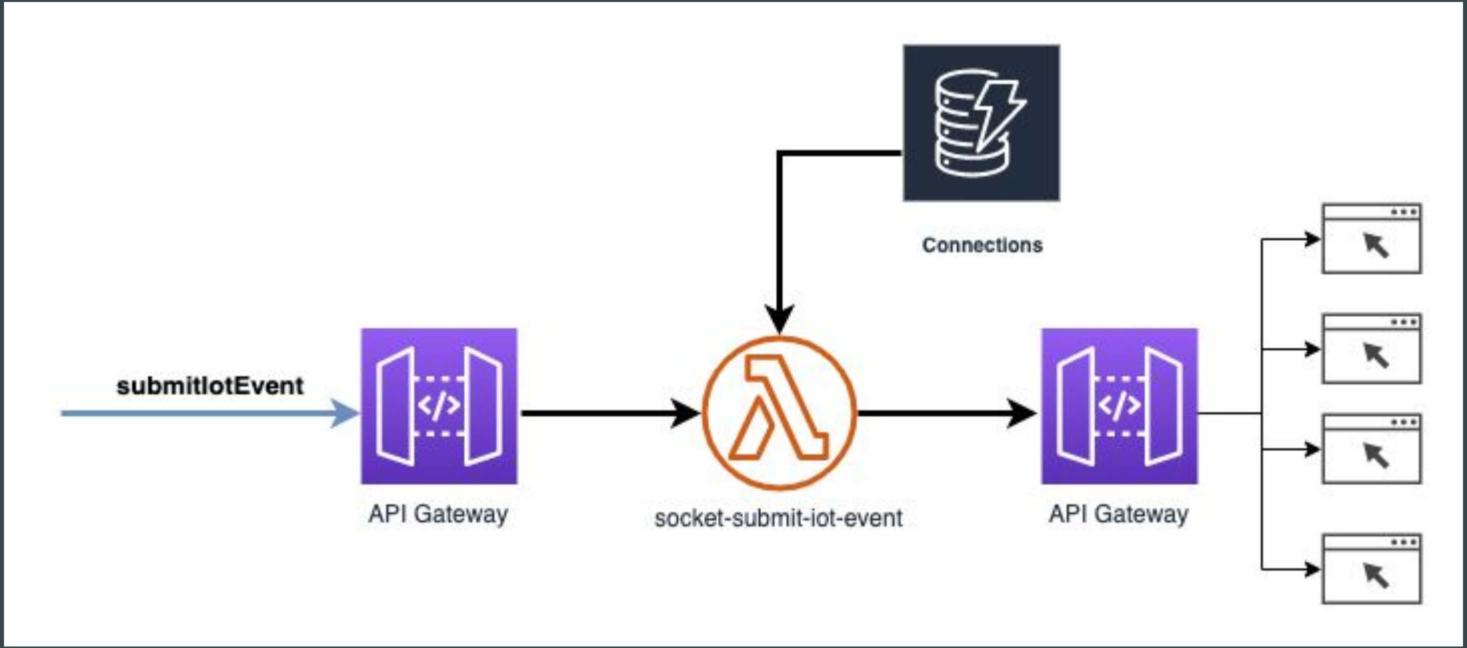
Workflow for submitting a new event.



```
{
  "action": "submitIotEvent",
  "event": {
    "MessageType": "NESClick",
    "Message": {
      "color": "blue",
      "button": "A",
      "axis": null,
      "value": null,
      "source": "websocket"
    }
  }
}
```



How do we send events to *specific* spectators?

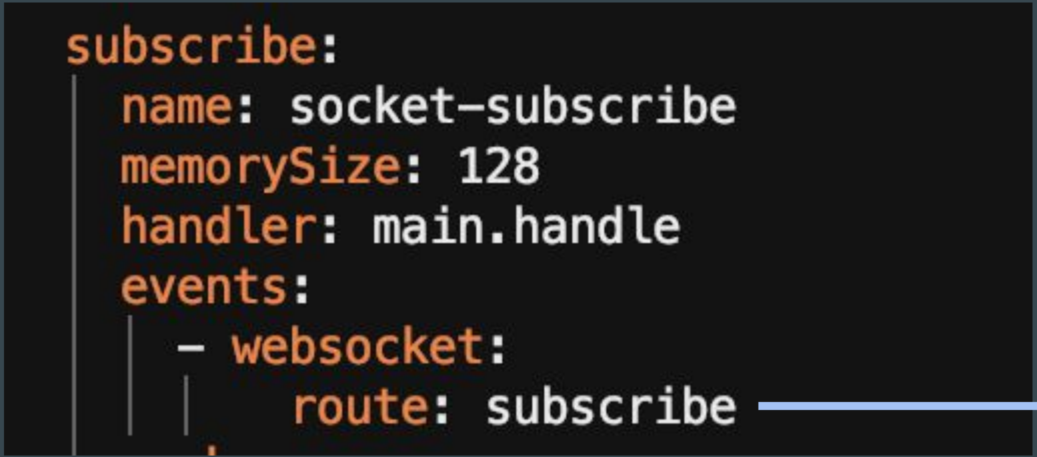


This is similar to how a chat client might work. The receiver might forward the message to the intended recipients based on what “room” they are in.

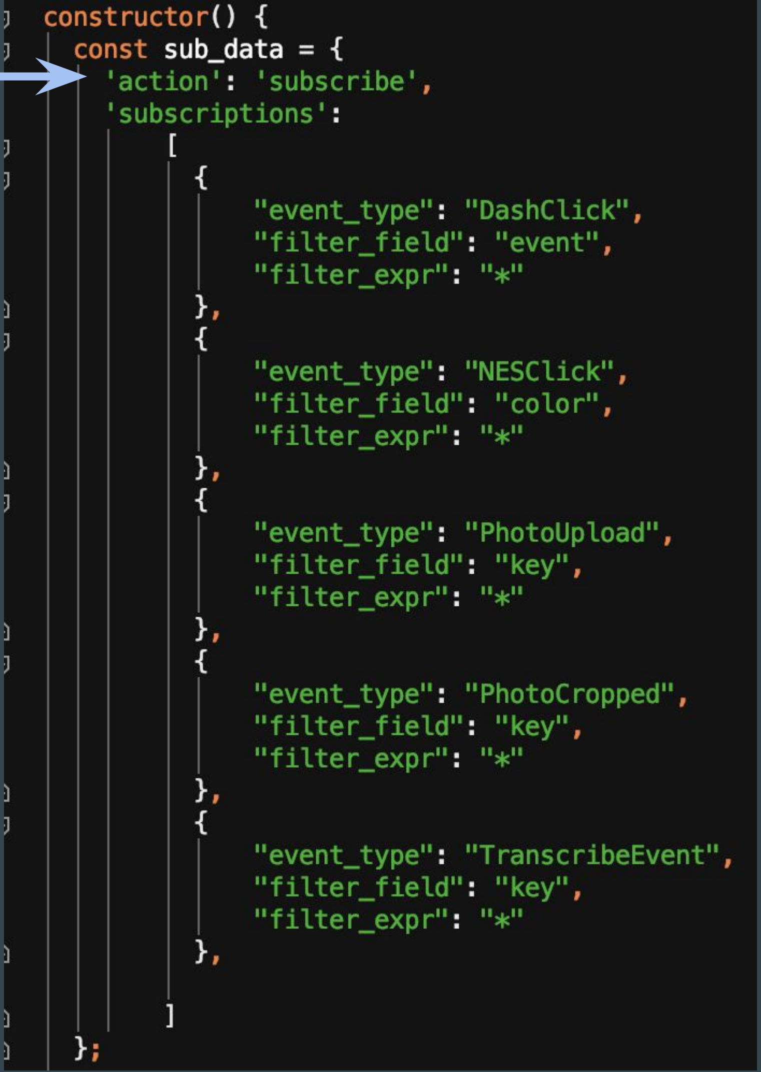
1) Spectators must subscribe.

- ‘subscribe’ route is created.

Spectators must select the types of events they are interested in by submitting a ‘subscribe’ message over the websocket.

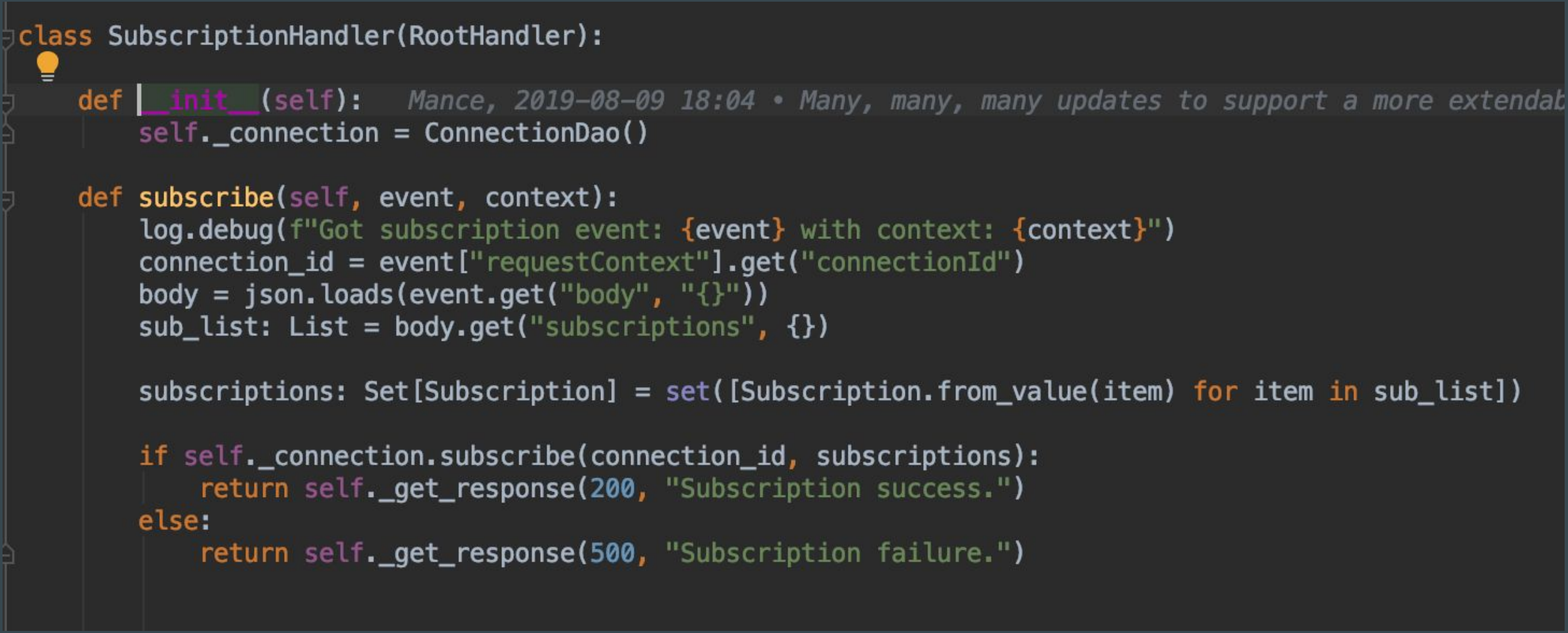


```
subscribe:
  name: socket-subscribe
  memorySize: 128
  handler: main.handle
  events:
    - websocket:
        route: subscribe
```



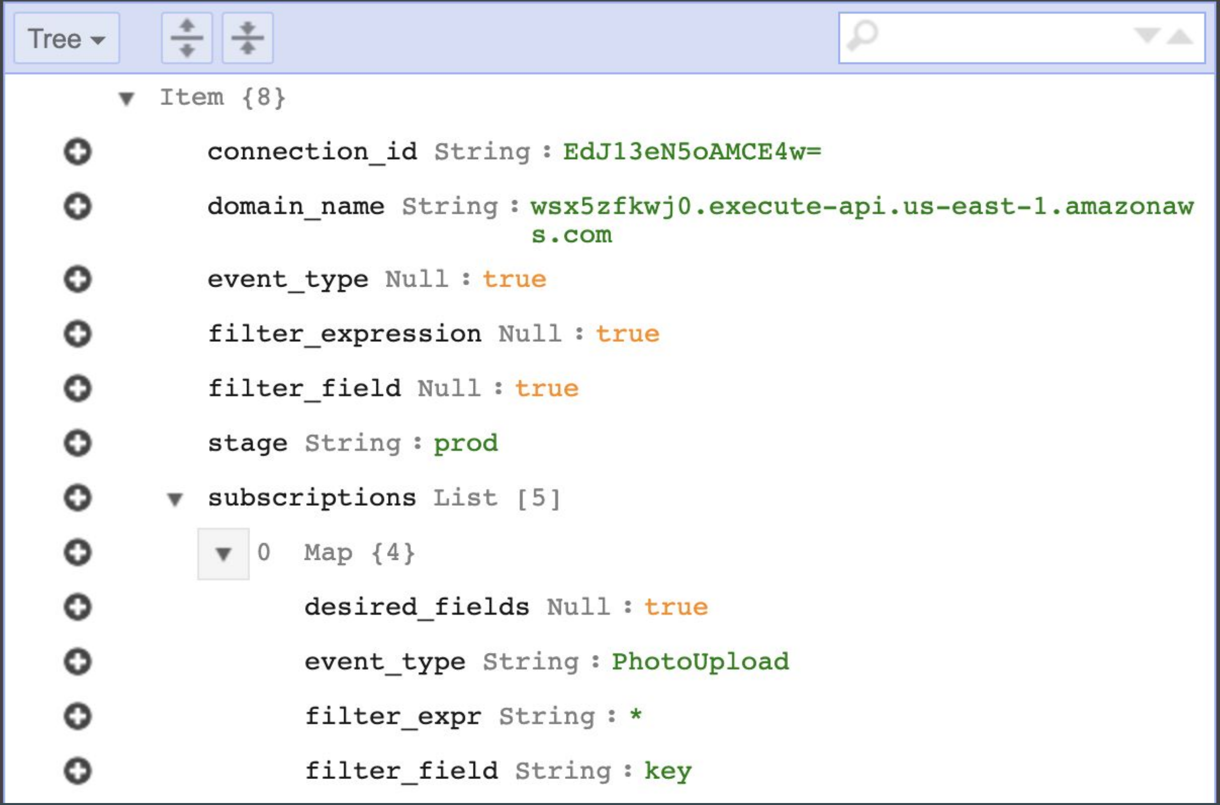
```
constructor() {
  const sub_data = {
    'action': 'subscribe',
    'subscriptions':
        [
          {
                "event_type": "DashClick",
                "filter_field": "event",
                "filter_expr": "*"
          },
          {
                "event_type": "NESClick",
                "filter_field": "color",
                "filter_expr": "*"
          },
          {
                "event_type": "PhotoUpload",
                "filter_field": "key",
                "filter_expr": "*"
          },
          {
                "event_type": "PhotoCropped",
                "filter_field": "key",
                "filter_expr": "*"
          },
          {
                "event_type": "TranscribeEvent",
                "filter_field": "key",
                "filter_expr": "*"
          },
        ]
  };
```

Subscription Handler



```
class SubscriptionHandler(RootHandler):

    def __init__(self):    Mance, 2019-08-09 18:04 • Many, many, many updates to support a more extendab
        self._connection = ConnectionDao()

    def subscribe(self, event, context):
        log.debug(f"Got subscription event: {event} with context: {context}")
        connection_id = event["requestContext"].get("connectionId")
        body = json.loads(event.get("body", "{}"))
        sub_list: List = body.get("subscriptions", {})

        subscriptions: Set[Subscription] = set([Subscription.from_value(item) for item in sub_list])

        if self._connection.subscribe(connection_id, subscriptions):
            return self._get_response(200, "Subscription success.")
        else:
            return self._get_response(500, "Subscription failure.")
```

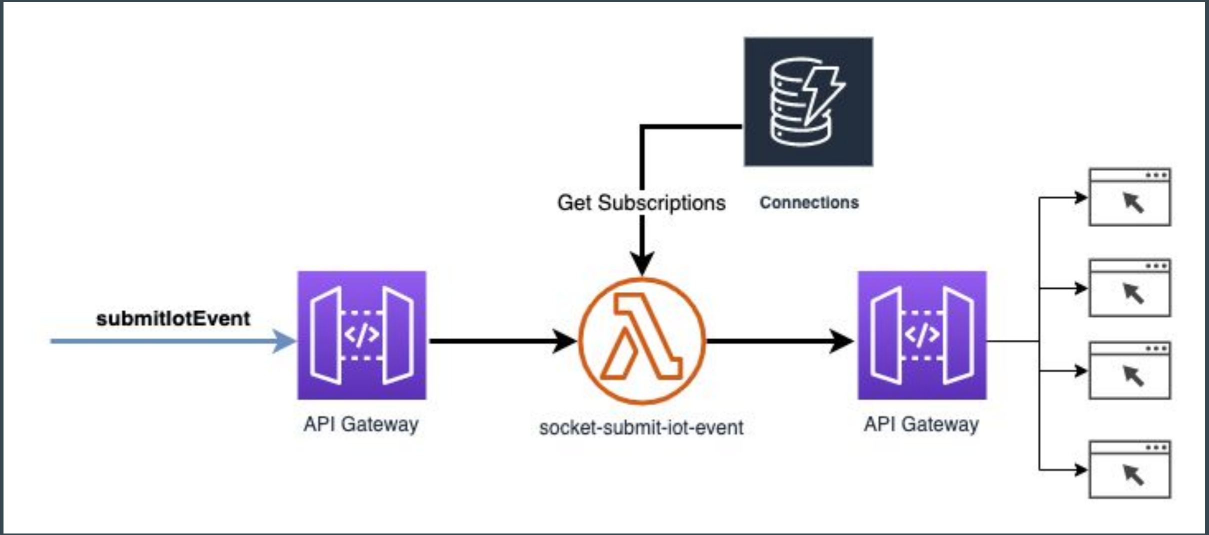
Subscriptions are saved to the DB with the connection_id



```
▼ Item {8}
    connection_id String : EdJ13eN5oAMCE4w=
    domain_name String : wsx5zfkwj0.execute-api.us-east-1.amazonaw
                         s.com
    event_type Null : true
    filter_expression Null : true
    filter_field Null : true
    stage String : prod
  ▼ subscriptions List [5]
    ▼ 0  Map {4}
        desired_fields Null : true
        event_type String : PhotoUpload
        filter_expr String : *
        filter_field String : key
```

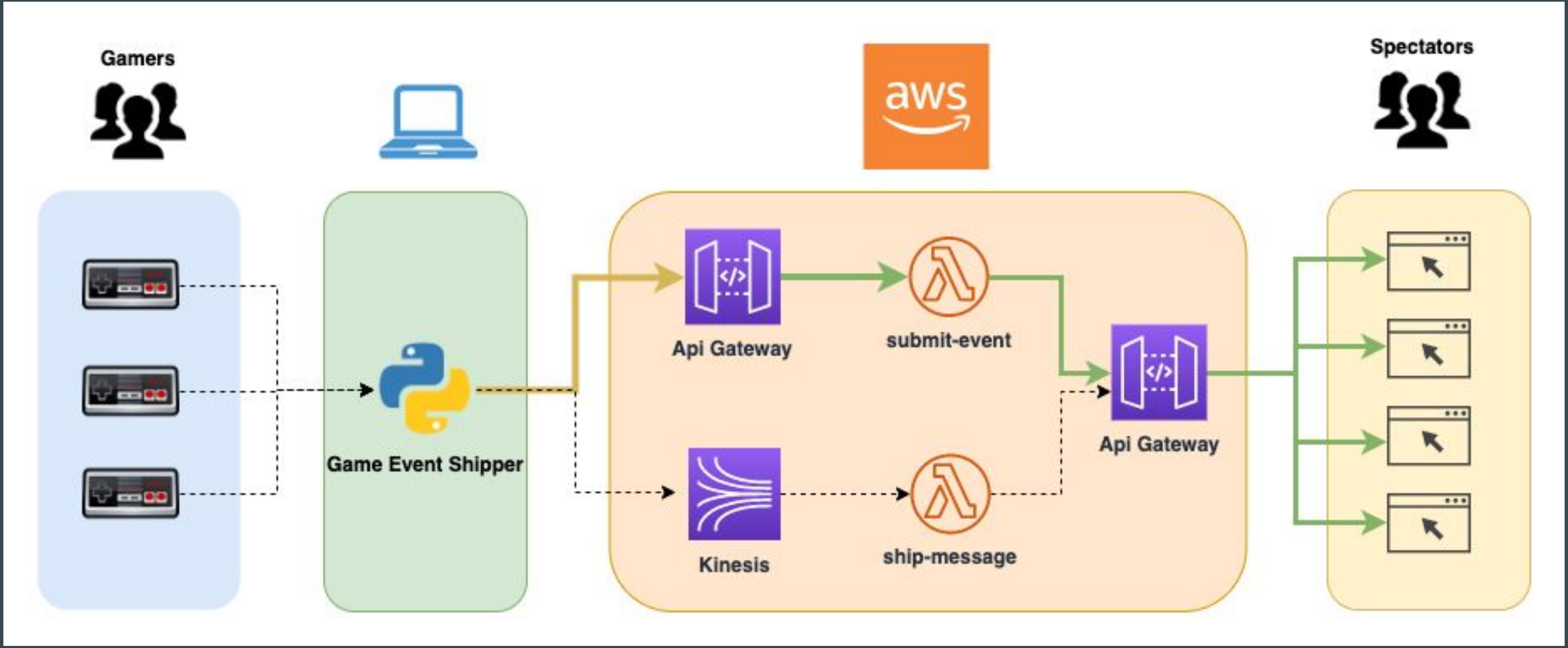
2) Compare messages to subscriptions + ship

- Subscribers submit their subscriptions to the game(s) they wish to view.
- The iot event processor lambda sends processed events to open websocket connections with matching subscriptions.



Event subscription demo.

How do we push events over the WebSocket?



Shipping Events

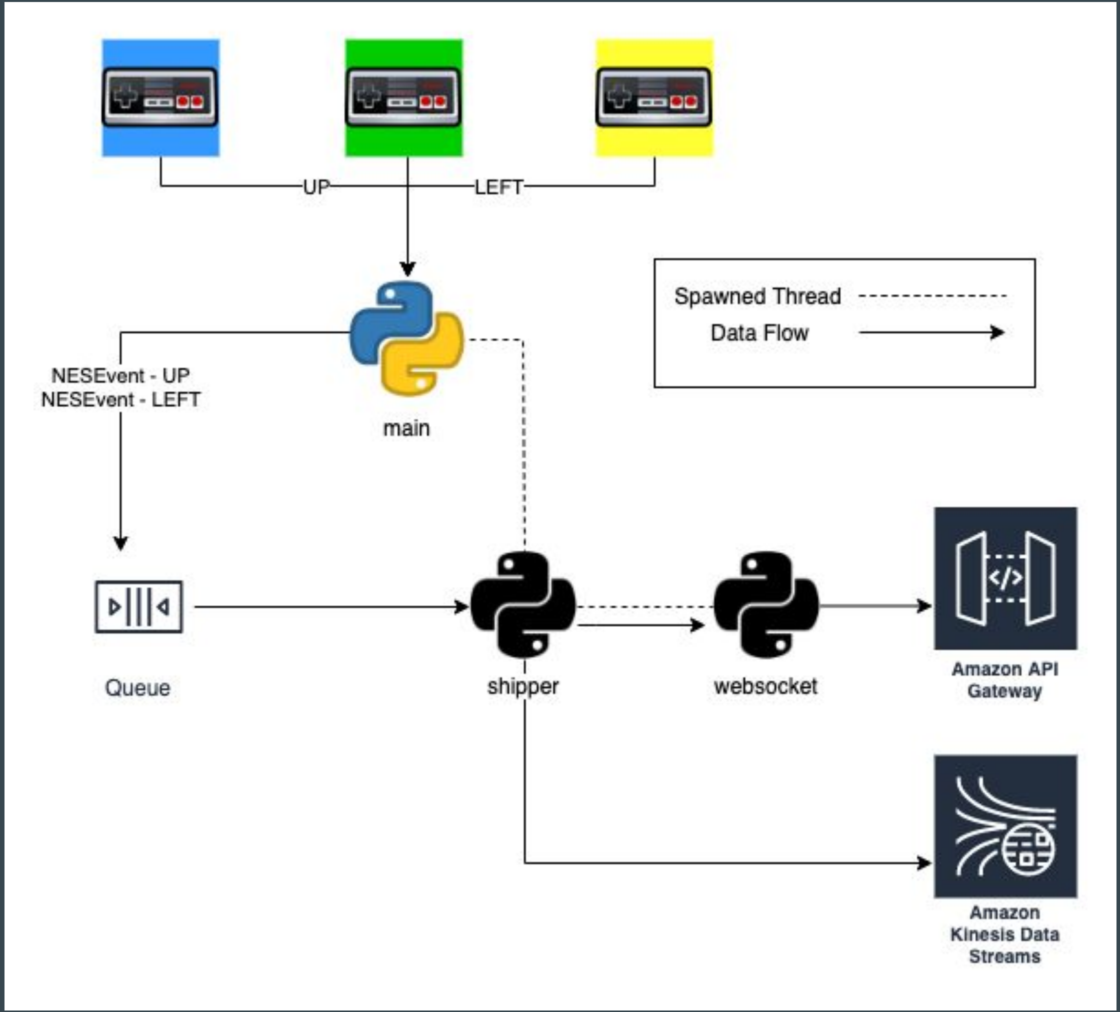
Main Process:

Listens 100 times/sec for input. Maps joystick inputs to “NESEvent” objects.

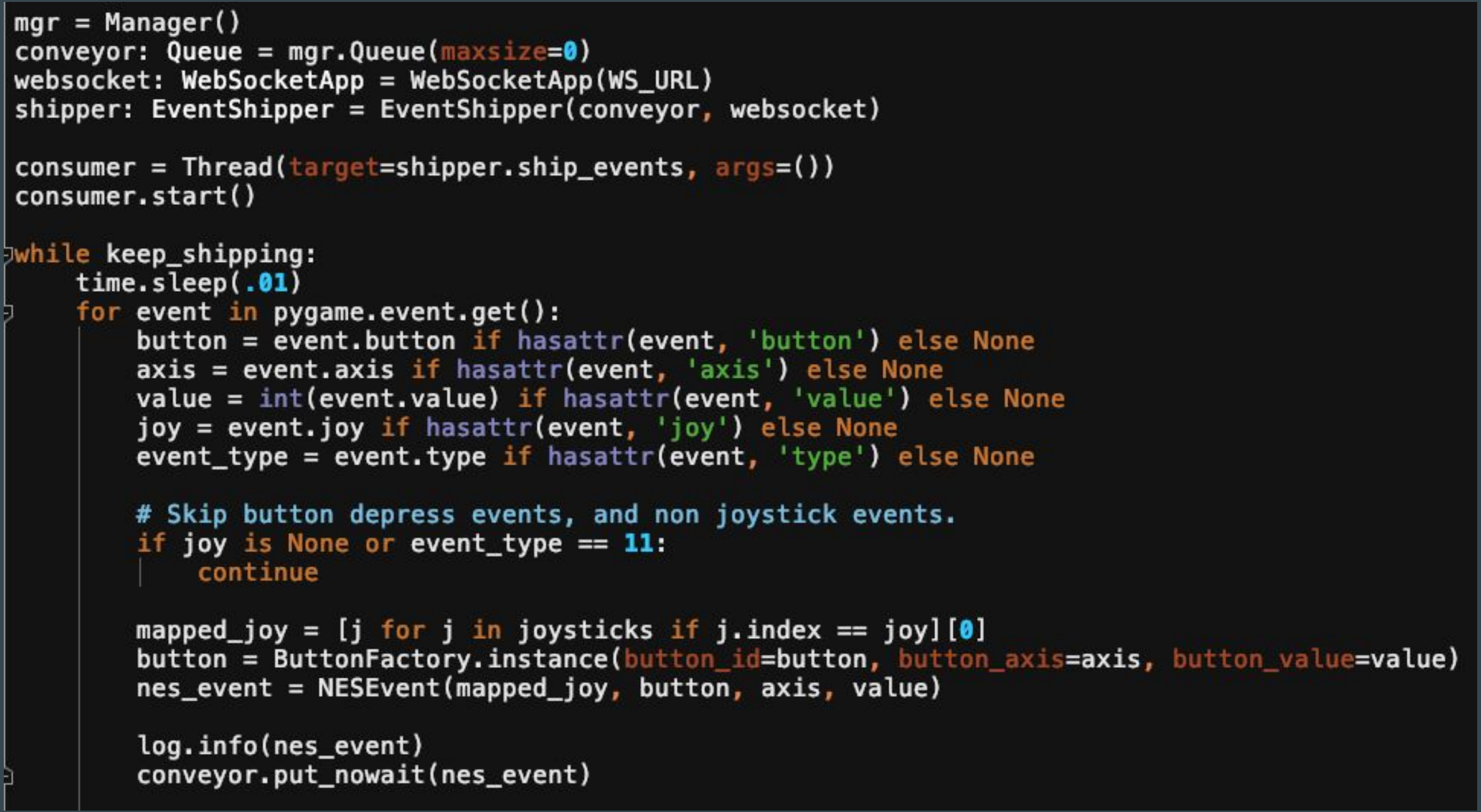
NESEvents are put on an in-memory queue.

Shipper Thread:

Pulls NESEvent messages from queue and pushes via WebSocket to API Gateway and to Kinesis over HTTP.

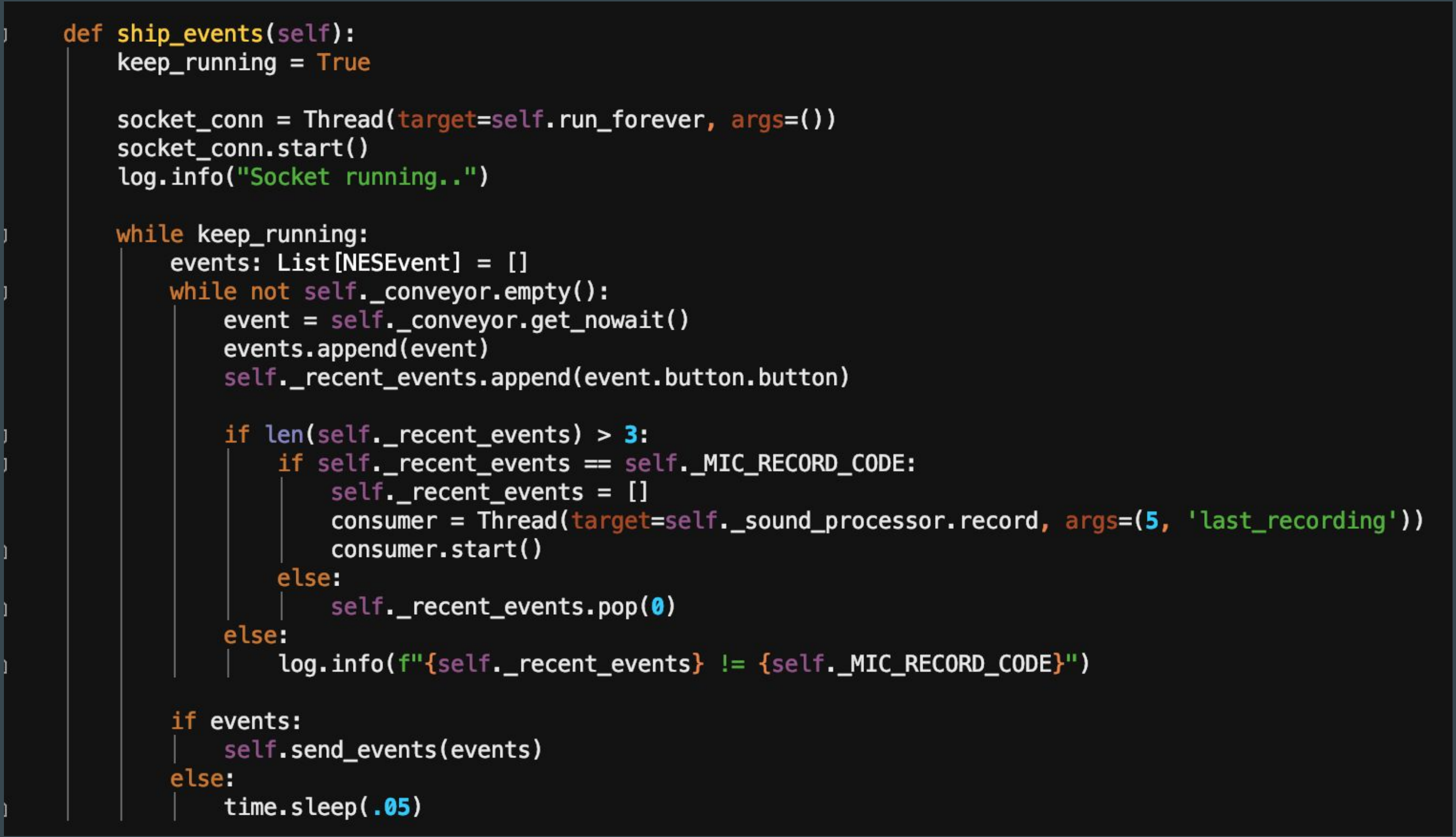


Receive + Queue Events



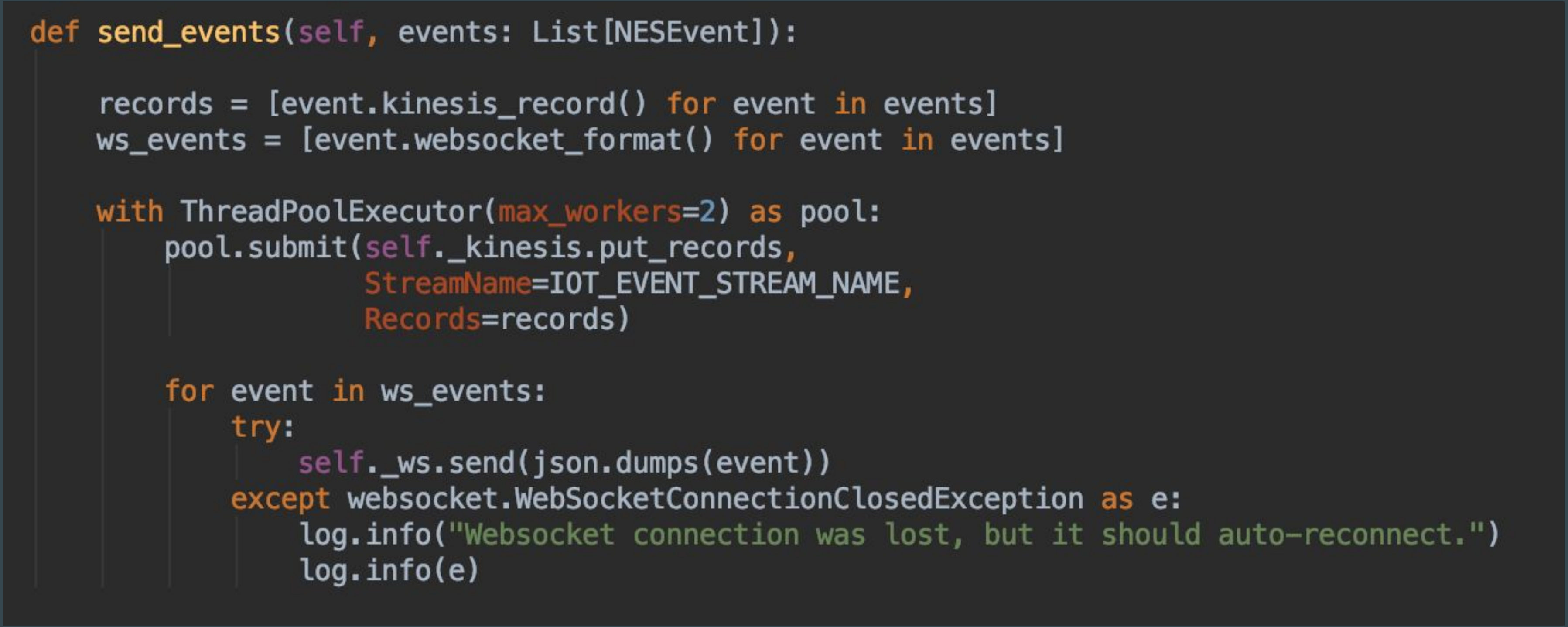
```python
mgr = Manager()
conveyor: Queue = mgr.Queue(maxsize=0)
websocket: WebSocketApp = WebSocketApp(WS_URL)
shipper: EventShipper = EventShipper(conveyor, websocket)

consumer = Thread(target=shipper.ship_events, args=())
consumer.start()

while keep_shipping:
    time.sleep(.01)
    for event in pygame.event.get():
        button = event.button if hasattr(event, 'button') else None
        axis = event.axis if hasattr(event, 'axis') else None
        value = int(event.value) if hasattr(event, 'value') else None
        joy = event.joy if hasattr(event, 'joy') else None
        event_type = event.type if hasattr(event, 'type') else None

        # Skip button depress events, and non joystick events.
        if joy is None or event_type == 11:
            continue

        mapped_joy = [j for j in joysticks if j.index == joy][0]
        button = ButtonFactory.instance(button_id=button, button_axis=axis, button_value=value)
        nes_event = NESEvent(mapped_joy, button, axis, value)

        log.info(nes_event)
        conveyor.put_nowait(nes_event)
```

Pull from Queue + Send



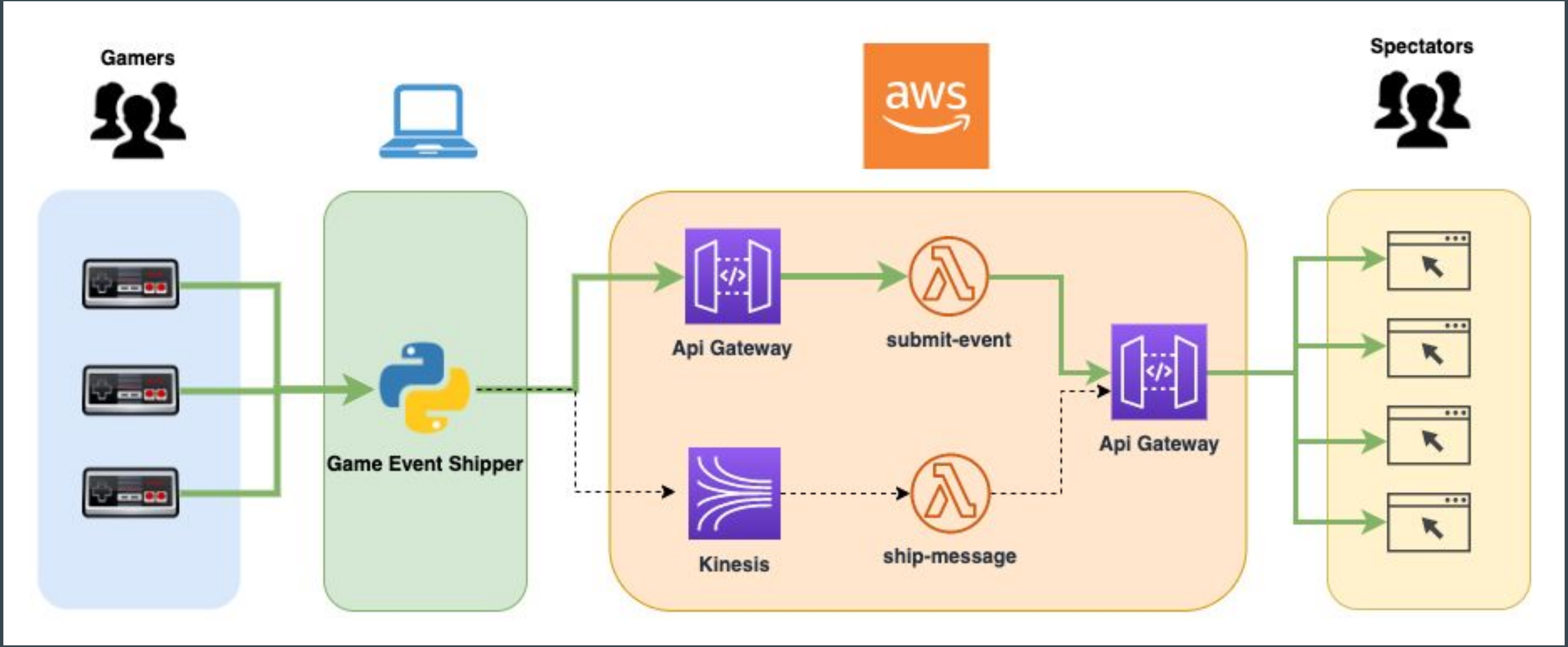
```python
def ship_events(self):
    keep_running = True

    socket_conn = Thread(target=self.run_forever, args=())
    socket_conn.start()
    log.info("Socket running..")

    while keep_running:
        events: List[NESEvent] = []
        while not self._conveyor.empty():
            event = self._conveyor.get_nowait()
            events.append(event)
            self._recent_events.append(event.button.button)

            if len(self._recent_events) > 3:
                if self._recent_events == self._MIC_RECORD_CODE:
                    self._recent_events = []
                    consumer = Thread(target=self._sound_processor.record, args=(5, 'last_recording'))
                    consumer.start()
                else:
                    self._recent_events.pop(0)
            else:
                log.info(f"{self._recent_events} != {self._MIC_RECORD_CODE}")

        if events:
            self.send_events(events)
        else:
            time.sleep(.05)
```

Ship to Kinesis + Websocket

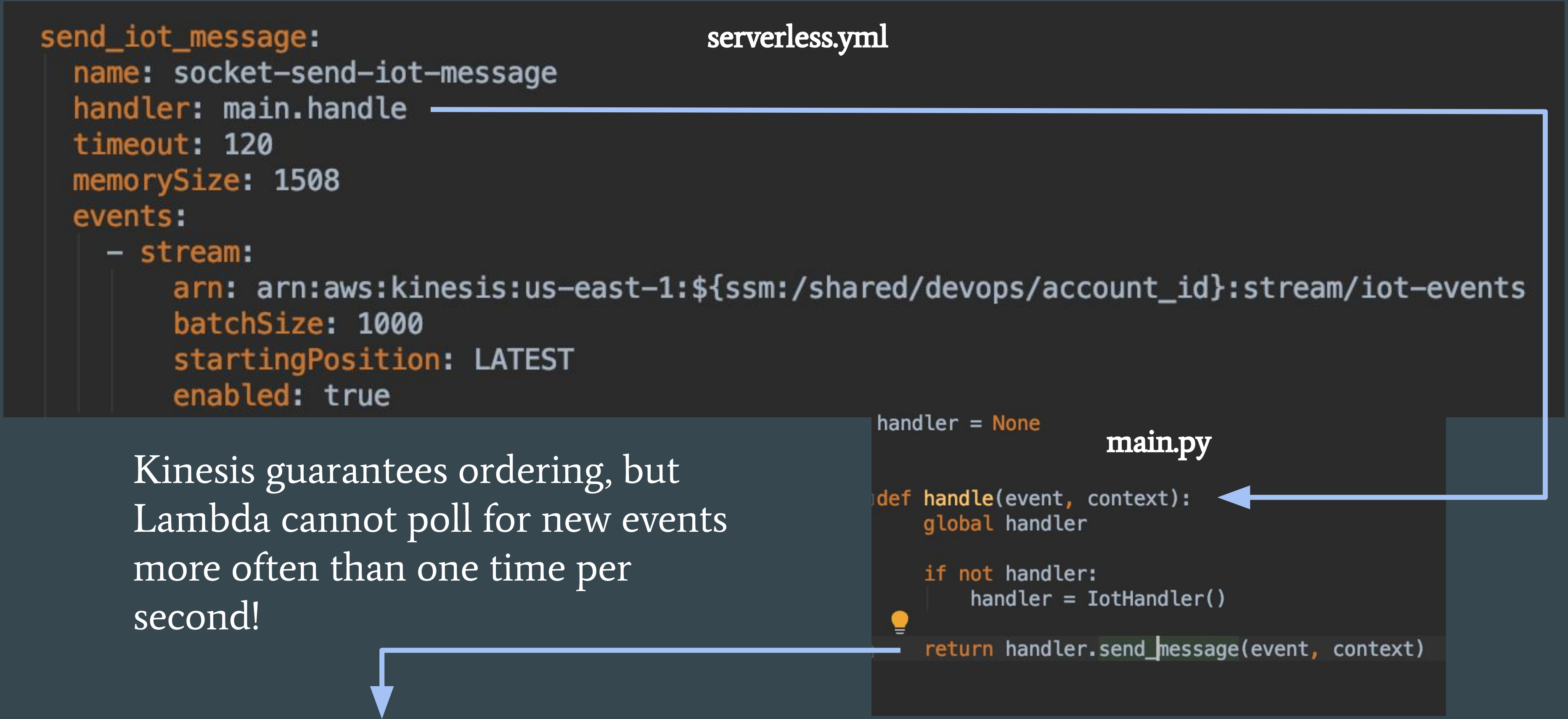


```python
def send_events(self, events: List[NESEvent]):

    records = [event.kinesis_record() for event in events]
    ws_events = [event.websocket_format() for event in events]

    with ThreadPoolExecutor(max_workers=2) as pool:
        pool.submit(self._kinesis.put_records,
                    StreamName=IOT_EVENT_STREAM_NAME,
                    Records=records)

        for event in ws_events:
            try:
                self._ws.send(json.dumps(event))
            except websocket.WebSocketConnectionClosedException as e:
                log.info("Websocket connection was lost, but it should auto-reconnect.")
                log.info(e)
```

What about processing Kinesis events?



Serverless framework supports Kinesis datasources!



serverless.yml

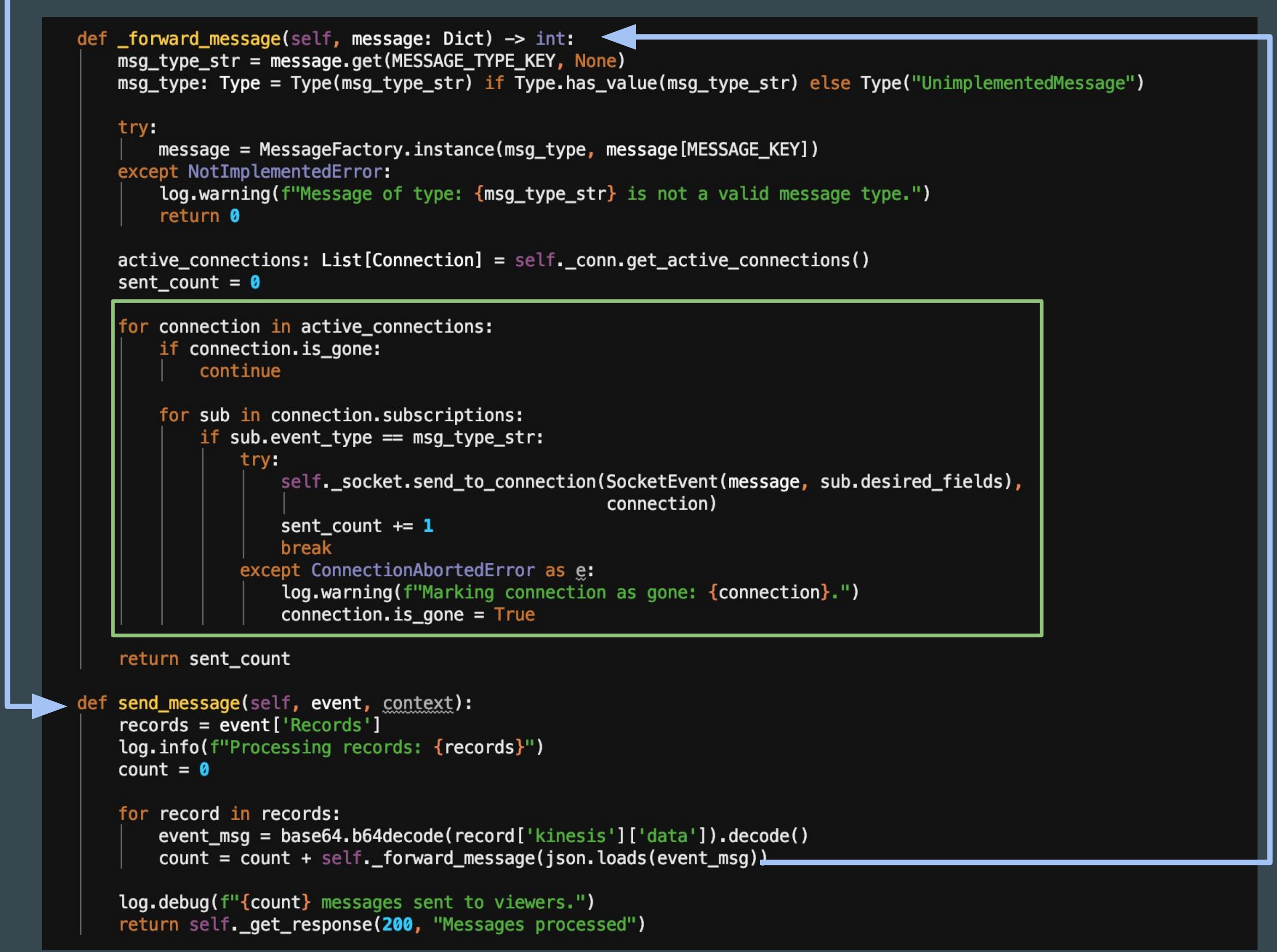
```yaml
send_iot_message:
  name: socket-send-iot-message
  handler: main.handle
  timeout: 120
  memorySize: 1508
  events:
    - stream:
        arn: arn:aws:kinesis:us-east-1:${ssm:/shared/devops/account_id}:stream/iot-events
        batchSize: 1000
        startingPosition: LATEST
        enabled: true
```

main.py

```python
handler = None


def handle(event, context):
    global handler

    if not handler:
        handler = IotHandler()

    return handler.send_message(event, context)
```

Kinesis guarantees ordering, but Lambda cannot poll for new events more often than one time per second!

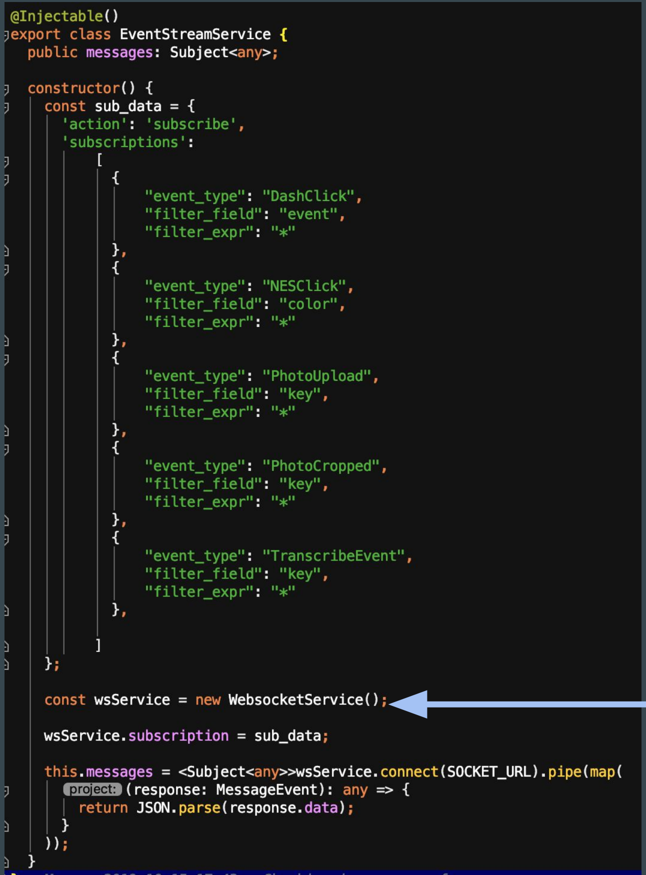
Forward messages to relevant subscribers.



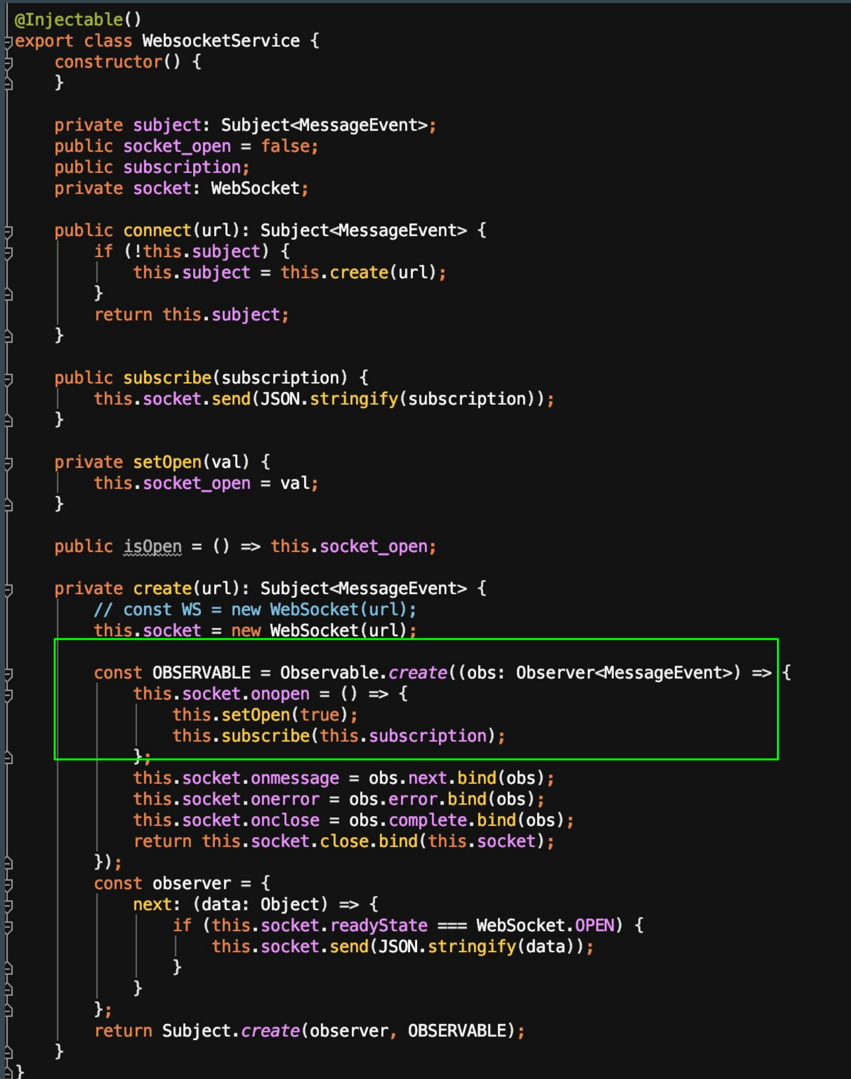
```python
def _forward_message(self, message: Dict) -> int:
    msg_type_str = message.get(MESSAGE_TYPE_KEY, None)
    msg_type: Type = Type(msg_type_str) if Type.has_value(msg_type_str) else Type("UnimplementedMessage")

    try:
        message = MessageFactory.instance(msg_type, message[MESSAGE_KEY])
    except NotImplementedError:
        log.warning(f"Message of type: {msg_type_str} is not a valid message type.")
        return 0

    active_connections: List[Connection] = self._conn.get_active_connections()
    sent_count = 0

    for connection in active_connections:
        if connection.is_gone:
            continue

        for sub in connection.subscriptions:
            if sub.event_type == msg_type_str:
                try:
                    self._socket.send_to_connection(SocketEvent(message, sub.desired_fields),
                                                    connection)
                    sent_count += 1
                    break
                except ConnectionAbortedError as e:
                    log.warning(f"Marking connection as gone: {connection}.")
                    connection.is_gone = True

    return sent_count

def send_message(self, event, context):
    records = event['Records']
    log.info(f"Processing records: {records}")
    count = 0

    for record in records:
        event_msg = base64.b64decode(record['kinesis']['data']).decode()
        count = count + self._forward_message(json.loads(event_msg))

    log.debug(f"{count} messages sent to viewers.")
    return self._get_response(200, "Messages processed")
```

Kinesis VS Websocket Speed Demo! :D

Getting events to the spectators



```
@Injectable()
export class EventStreamService {
  public messages: Subject<any>;

  constructor() {
    const sub_data = {
      'action': 'subscribe',
      'subscriptions':
          [
            {
                "event_type": "DashClick",
                "filter_field": "event",
                "filter_expr": "*"
            },
            {
                "event_type": "NESClick",
                "filter_field": "color",
                "filter_expr": "*"
            },
            {
                "event_type": "PhotoUpload",
                "filter_field": "key",
                "filter_expr": "*"
            },
            {
                "event_type": "PhotoCropped",
                "filter_field": "key",
                "filter_expr": "*"
            },
            {
                "event_type": "TranscribeEvent",
                "filter_field": "key",
                "filter_expr": "*"
            },

          ]
    };

    const wsService = new WebsocketService();

    wsService.subscription = sub_data;

    this.messages = <Subject<any>>wsService.connect(SOCKET_URL).pipe(map(
      project: (response: MessageEvent): any => {
        return JSON.parse(response.data);
      }
    ));
  }
```

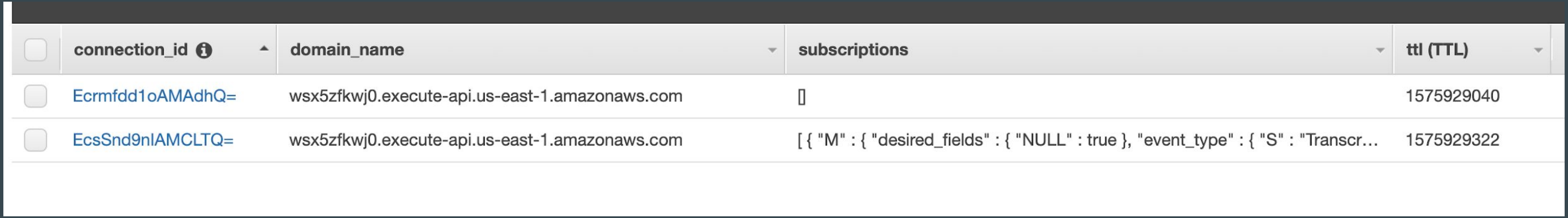


```
@Injectable()
export class WebsocketService {
    constructor() {
    }

    private subject: Subject<MessageEvent>;
    public socket_open = false;
    public subscription;
    private socket: WebSocket;

    public connect(url): Subject<MessageEvent> {
        if (!this.subject) {
            this.subject = this.create(url);
        }
        return this.subject;
    }

    public subscribe(subscription) {
        this.socket.send(JSON.stringify(subscription));
    }

    private setOpen(val) {
        this.socket_open = val;
    }

    public isOpen = () => this.socket_open;

    private create(url): Subject<MessageEvent> {
        // const WS = new WebSocket(url);
        this.socket = new WebSocket(url);

        const OBSERVABLE = Observable.create((obs: Observer<MessageEvent>) => {
            this.socket.onopen = () => {
                this.setOpen(true);
                this.subscribe(this.subscription);
            };
            this.socket.onmessage = obs.next.bind(obs);
            this.socket.onerror = obs.error.bind(obs);
            this.socket.onclose = obs.complete.bind(obs);
            return this.socket.close.bind(this.socket);
        });
        const observer = {
            next: (data: Object) => {
                if (this.socket.readyState === WebSocket.OPEN) {
                    this.socket.send(JSON.stringify(data));
                }
            }
        };
        return Subject.create(observer, OBSERVABLE);
    }
}
```

Like before. Subscription is now in the DB

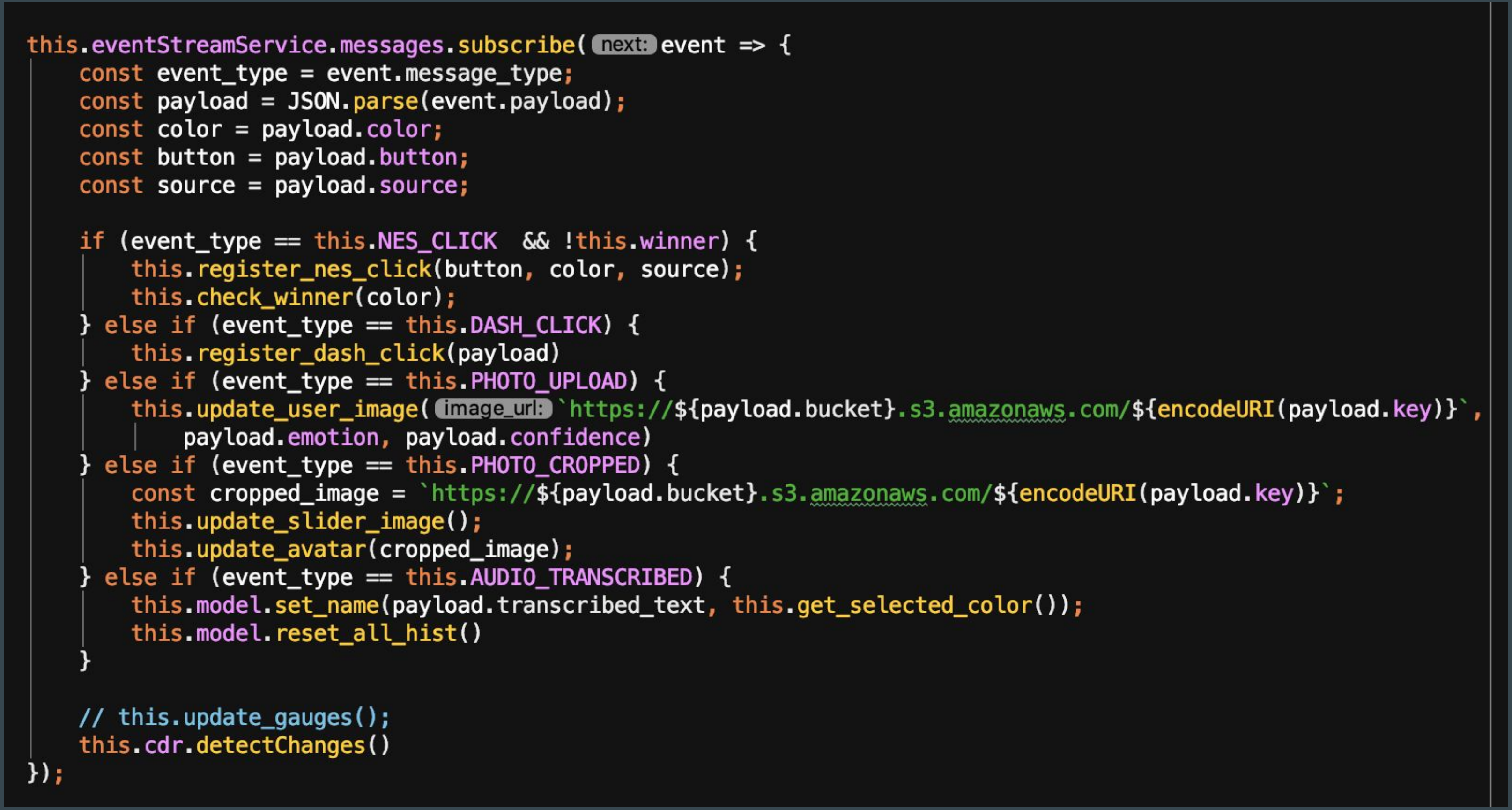
Maintains a list of all open WebSocket connections.

Includes both game VIEWERS and game event SHIPPERS



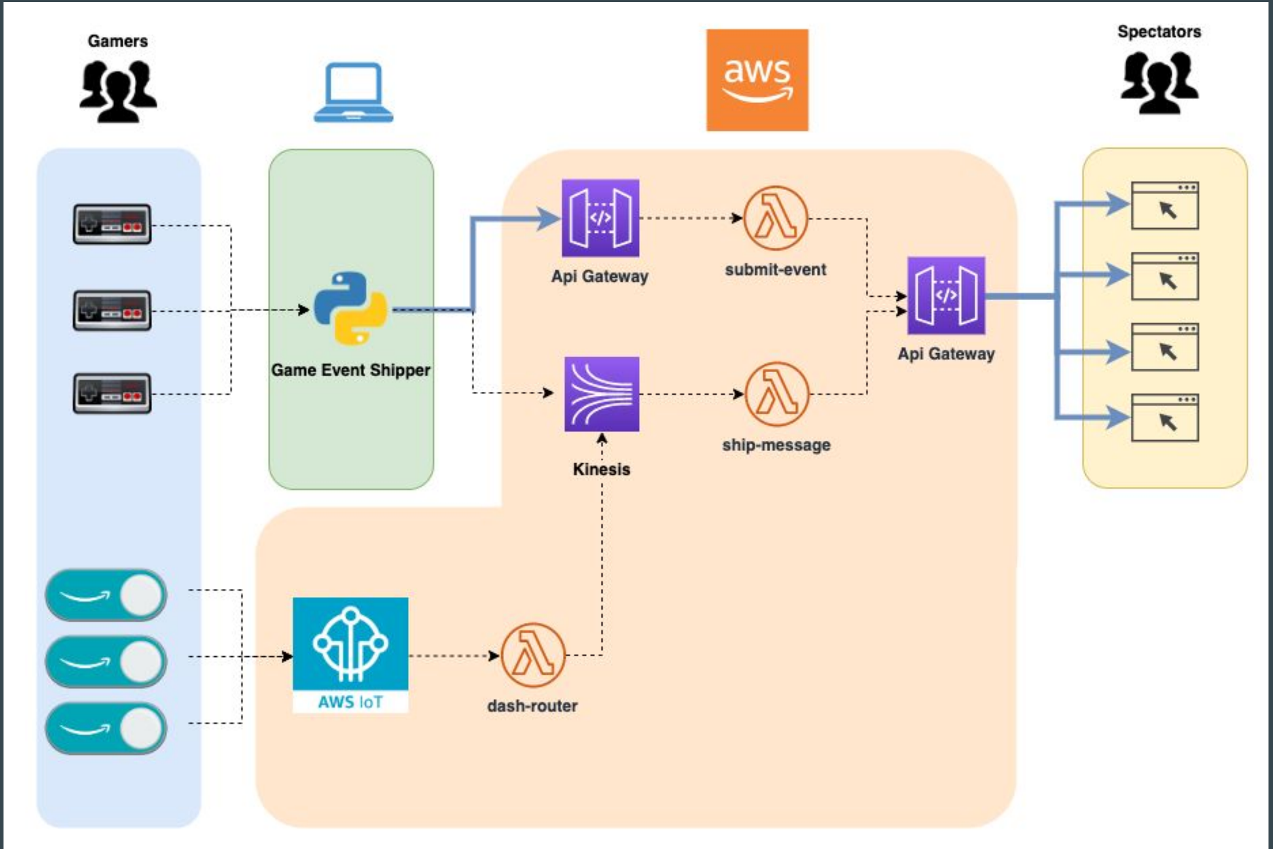
connection_id	domain_name	subscriptions	ttl (TTL)
Ecrmfdd1oAMAdhQ=	wsx5zfkwj0.execute-api.us-east-1.amazonaws.com	[]	1575929040
EcsSnd9nIAMCLTQ=	wsx5zfkwj0.execute-api.us-east-1.amazonaws.com	[{ "M" : { "desired_fields" : { "NULL" : true }, "event_type" : { "S" : "Transcr...	1575929322

Next, listen to events.



```
this.eventStreamService.messages.subscribe( next: event => {
    const event_type = event.message_type;
    const payload = JSON.parse(event.payload);
    const color = payload.color;
    const button = payload.button;
    const source = payload.source;

    if (event_type == this.NES_CLICK  && !this.winner) {
        this.register_nes_click(button, color, source);
        this.check_winner(color);
    } else if (event_type == this.DASH_CLICK) {
        this.register_dash_click(payload)
    } else if (event_type == this.PHOTO_UPLOAD) {
        this.update_user_image( image_url: `https://${payload.bucket}.s3.amazonaws.com/${encodeURI(payload.key)}`,
            payload.emotion, payload.confidence)
    } else if (event_type == this.PHOTO_CROPPED) {
        const cropped_image = `https://${payload.bucket}.s3.amazonaws.com/${encodeURI(payload.key)}`;
        this.update_slider_image();
        this.update_avatar(cropped_image);
    } else if (event_type == this.AUDIO_TRANSCRIBED) {
        this.model.set_name(payload.transcribed_text, this.get_selected_color());
        this.model.reset_all_hist()
    }

    // this.update_gauges();
    this.cdr.detectChanges()
});
```

State is stored in browser (now).



I’m tired of making more slides. Lets game!