Asynchronous architectures with AWS EventBridge

Agenda

- Scenario introduction
- Classic architecture challenges
- Patterns for improvement
- EventBridge concepts
- Asynchronous responses
- Other features
- Operational aspects
- Additional notes

Scenario introduction

Fakémon GO!

We are going to create an exciting **capture-creature game** for mobile devices.

Expecting tones of concurrent users interacting heavily with the server in complex ways, as it will be a **massive multiplayer** experience.

The situation is not very different from what we would find in an **e-commerce application** during peak times, with multiple **customers fighting for the last unit** of a particular product.

User story

- A fakémon appears in the capture reticle.
- The player makes an **attempt to capture it**.
- It can fail if another player is quicker or the fakémon dodges.
- If the attempt success, the player collection of fakémon must be updated and the world needs to reflect the change.

Classic architecture challenges

First architecture approach

Our familiar approach is supported by an *Application Load Balancer*, a fleet of servers and a *RDS* Postgres instance.

Draw the typical elb + asg + rds architecture.

Current architecture limitations

- The autoscaling group **doesn't respond quickly enough** to spikes, forcing us to over provisioning the system
- The database is **not able to keep up with the writes** unless we use a huge instance
- Different parts of the game **are coupled**, making it difficult to evolve the source code

Some of the problems are related to the fact that the lifecycle of the requests is synchronous, keeping resources locked during the whole time. Also, a single database engine is taking care of all the tasks related to representing the game state. Coupling happens when, for example, representing the current world for one player affects the performance of capturing a fakémon for another one.

Patterns for improvement

Software architecture

- The *application controller* is the component that interacts with the player.
- The *capture service* contains the logic to… well… capture the fakémon.

Procedure invocation component communication

This is the classic approach for application development.

The *application controller* instructs the *capture service* to execute the logic for capturing a fakémon.

It is, in the end, a remote invocation of an operation. This pattern generates a **tight coupling** between both components.

Event-driven component communication

The *capture service* is registered as interested in responding to capture attempts.

The *application controller* broadcasts that one of such attempts has happened.

None of the components are aware of each other, thus **avoiding coupling**. Communications are implemented by generating and capturing messages.

Both services are not coupled anymore: neither of them know what components are generating nor responding to the event.

Synchronous actions

The response to the player's actions only arrives once the entire process has been completed, including the parts that are not relevant for continuing with the game. The component that started the request will pause until the cycle finishes.

The flow of the application is **easy to follow** in the code, and no sophisticated language syntax support is required. But resources are **blocked for a longer period** of time, and **latency is bigger** as a consequence of that waiting.

In other words, a component asks another one "Please, do this right now", and then waits for the response.

Asynchronous actions

Actions happen independently, without the *capture service* component blocking the *application controller*.

Once the task has finished, all the interested actors receive a **notification of the result**.

In this case, one component asks to others "whenever you can, please do it". Whenever the action is finished, the source component is notified of the result.

Fan-out pattern

Once the *capture service* generates the signal that the attempt to capture a fakémon succeeded, subscribed components may react to it **in parallel by getting their own copy** of the relevant information.

Splitting the flow in several parallel tasks is implicit, instead of part of the application's code.

EventBridge concepts

Event

A description of **what has happened**, in *json* format.

```
{
    "version": "0",
    "id": "910e8b4d-a55b-89cf-52a7-27854373da0f",
    "detail-type": "fakemon.capture_attempt",
    "source": "fakemon.attempt-to-capture-api",
    "account": "997122619583",
    "time": "2024-05-14T07:42:21Z",
    "region": "eu-west-1",
    "resources": [],
    "detail": {
        "player": "Alice",
        "pokemon": "fire-squirrel-32xa"
    }
}
```

Event bus

A **serverless** event ingestion mechanism that receives the events.

There is a **default event bus** used by AWS for providing details about what is happening with the infrastructure. There are several **third-party integrations** for connecting external tools (like DataDog, Okta, etc) with AWS. Applications can make use of their own **custom event buses**.

API gateway integration

The implementation will provide a very simple way of putting an event in the service bus as a response to an HTTPs request **without any additional code needed**.

The application will invoke this API when the player indicates their attempt to capture a fakémon, and the integration will put a `fakemon.capture_attempt` event in the bus.

Targets

A **copy of the event** will be delivered to each registered target.

A *lambda function* will be the target of the `capture_attempt` event, implementing the actual fakémon catch logic and generating a `capture_succeeded` or `capture_failed` event.

The lambda function will require a high-performance storage, so we suggest to implement the capturing state management using DynamoDB.

Here there is the list of available targets, although by using API destinations the actual event receivers can be any URL.

Rules

A **filter** that sends a **copy of an event** to a required **target**. They are simple to write, replicating the structure of the message.

If necessary, EventBridge **can apply transformations** before the copy of the event is delivered to the target.

```
{
  "detail-type": ["fakemon.capture_attempt"]
}
```

There is a simpler way of configuring end-to-end communications between several sources and a single destination, called pipes

Asynchronous responses

How the client receives the results?

- **Webhooks** (API destination endpoints) are an option only for very stable sources.
- Polling from **queues** can be leveraged in many scenarios, but typically incur in additional latency.
- **TCP connections** can be an efficient solution and it is available in many different technology stacks.

Event-based asynchronous architectures are disconnected by nature, so the action of starting a workflow should be considered a *fire-and-forget* situation.

Webhooks are not usually a good solution, as most endpoints are dynamic, private or placed behind a firewall.

If the number of targets are restricted, it is usually possible to use independent queues to hold the returning messages.

Otherwise, it is easier to keep a callback connection. Websockets are the natural fit if the application front-end is based on web technology.

Websockets

It is a well-established web technology for supporting stable TCP connections, based on **upgrading a HTTP** one. They are popular even in mobile applications, given the excellent support provided by popular servers.

It has better support than Server-sent events.

Will be replaced by WebTransport, but not today.

Websockets are indeed stable connections, but usually both ends of the wire are suspended in idle state most of the time (at least for this kind of architectures). That condition makes them very efficient in resource terms.

API gateway websocket connection

There is a **specific type of API** for this separated from the HTTP and REST kinds.

Once a new connection is created by invoking the $connect , the API gateway will call a *lambda function* that can extract the player name from the *request body* and store it associated with a provided *connection id*.

API gateway websocket response

Once the result of a workflow is available, the APIgw management API can be leveraged to send information through the corresponding connection.

This management API can be invoked by a *lambda function* **once the response is ready** (when the attempt of capturing the fakémon has succeeded or failed).

It is possible to get the connection id using `curl`, as described in Use @connections command in your backend service.

Other features (sneak peek)

Scheduler

Using the default bus, it is possible to invoke targets based on *cron* expressions, or at a particular rate.

This is an extremely convenient **serverless cron**.

Schema registry

It is a service providing a **catalog of the structure** that events are expected to have.

This is a critical feature required to apply **governance** and documentation in any complex system.

Dead letter queues

Events not processed by their targets can be redirected to Dead Letter Queues for **further analysis** (and forwarded again to the corresponding event bus if needed).

Archive and replay

EventBridge supports **storing** a subset (or all) the events and **replay** them again, making it possible (up to a certain point) recreating the current state of the system.

It doesn't provide good enough time granularity to implement this reconstruction in a simple way, neither it is practical to do it if third-party elements are in place.

Operational aspects

Monitoring

Most useful CloudWatch metrics:

- *Events*: Count of partner events ingested.
- *FailedInvocations*: The number of invocations that failed permanently.
- *IngestiontoInvocationCompleteLatency*: Time taken from event ingestion to completion of the first successful invocation attempt.
- *ThrottledRules*: Number of times rule execution was throttled.

Debugging

It is very useful to add an additional rule **capturing all events** in the account/workflow/component and write them to a **CloudWatch Log stream**.

There is a direct integration for configuring EventBridge to do it.

Additional notes

Conclusions

Asynchronous, event-based architectures are **not so complicated** and have some advantages in performance and efficiency.

EventBridge provides a **serverless** platform for deploying them.

Default bus provides an awesome way of creating almost **real-time remediation** to many common problems.

Debugging and governing this kind of decoupled architectures is a real challenge.

Further readings

- EventBridge documentation
- Sample application
- Terraform module
- Serverless Framework support

Clap if you enjoyed it!

Javi Moreno

linkedin.com/in/javier-more

Breaking news: Hands-on workshop in preparation!