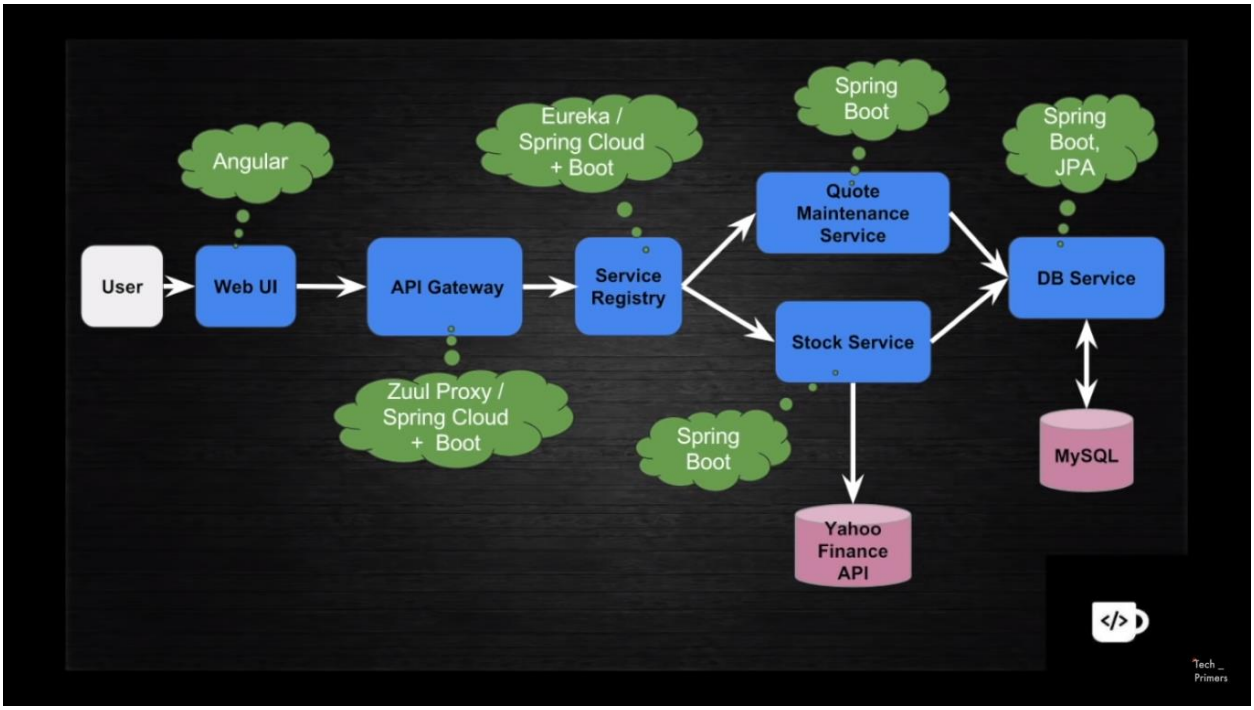
Micro Services:



1. **How Spring boot runs tomcat server when we run main method.**

   Maven downloads tomcat libraries and call the tomcat api to get the tomcat instance.

2. **How to change the server port in Spring Boot**

   Server.port in application.properties

3. **do I need main method if I develop web app as war using Spring Boot?**

Main method is not required for the typical deployment scenario of building a war and placing it in webapps folder of Tomcat. All you need is:

```
public class Application extends SpringBootServletInitializer {
        @Override
        protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
                return application.sources(Application.class);
        }
}
```

4. **Is @Bean lazy or eager Intialized**

By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is not desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized. A lazy-

initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

5. What is Rest Template

Used to communicate between rest api, new version is WebClient. . WebClient is alternative for Rest Template. Rest template is depriciated in Spring 5

6. WebClient

Web Client is to communicate API to Api through rest services. It's a alternative for Rest Template. Rest template is depriciated in Spring 5

```
Movie movie = webClientBuilder.build() WebClient
        .get() RequestHeadersUriSpec<capture of ?>
        .uri( s: "http://localhost:8082/movies/" + rating.getMovieId()) capture of
        .retrieve() ResponseSpec
        .bodyToMono(Movie.class) Mono<Movie>
        .block();
```

7. How Microservice communicate each other.

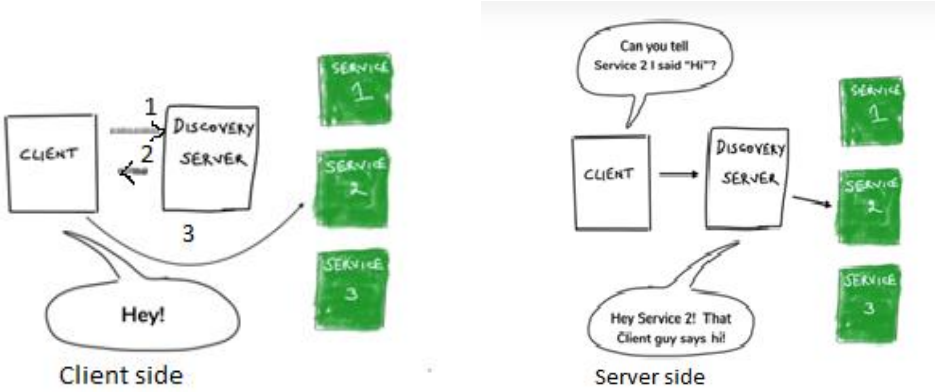
   Through Res api using WebClient or Rest Template.

8. **What is service descovery**

   Maintaining dynamic URL, cloud has dynamic urls genrated
   Load Balanced
   Multiple environment maintennce
   Eg: Eureka
   @EnableEurekaServer
   @EnableEurekaClient

Service Discovery
/ \

Client Side Server Side



9. How to do load balancing in Eureka

@LoadBalanced annotation does this. Load balancing can be from client side or server side.

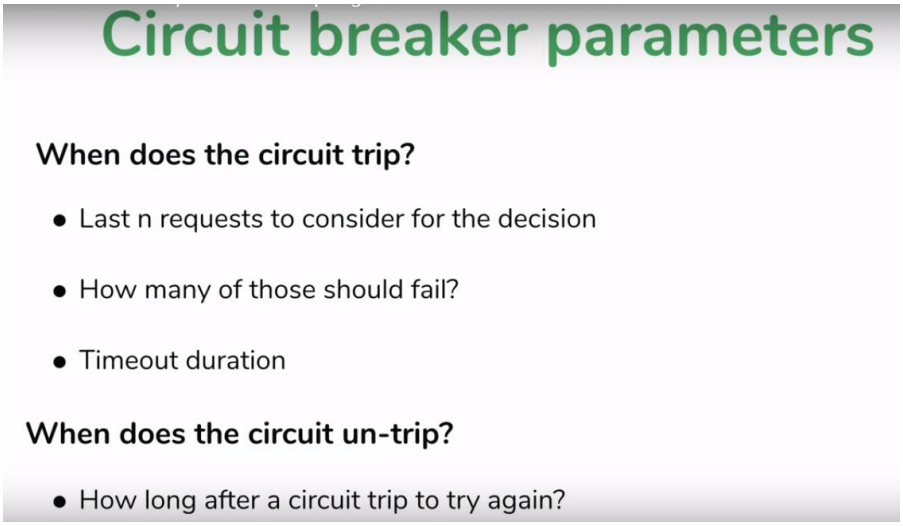
10. Issues with micro services

Scenrio: Micor service can go down
Solution: Have multiple service instance
Scenrio: Instance is slow
Solution: 1. Time out
2. Circuit breaker pattern – Hystrix
3. BulkHead pattern

11. Still slow? Circuit Breaker pattern.

- Detect something is wrong
- Take temporary steps to avoid the situation getting worse
- Deactivate the "problem" component so that it doesn't affect downstream components

12. **Circuit breaker parameters**

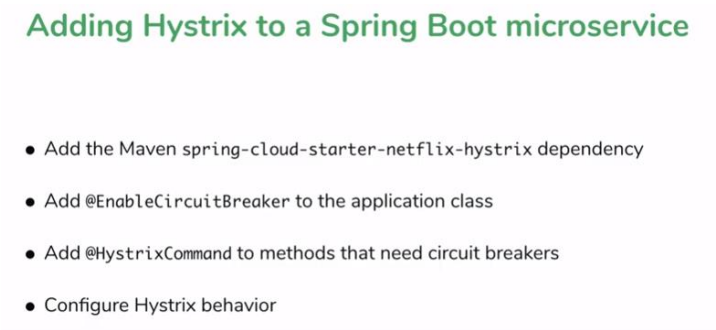


13. What happens when the circuit is break down?

Ans: fallback

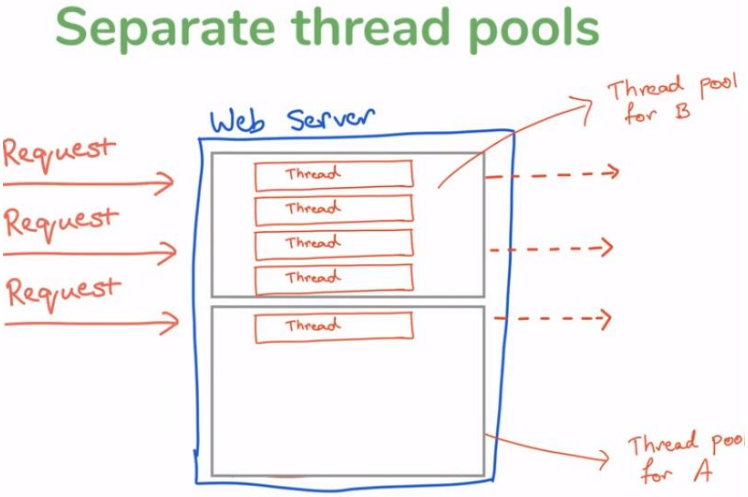
- Throw an error
- Return a fallback "default" response
- Save previous responses (cache) and use that when possible

14. Steps to implement Hystrix



15. How Hystrix works:
    Ans Proxy, it works proxy instance that handles circuit breaker pattern

16. Bulkhead pattern: Separate thread pool for each type of microservices.



Service Mesh, Service Discovery and API Gateways

CASE 1: WHAT ABOUT MICROSERVICES?

In case you're in a service architecture it is possible that, according to your boundary strategy, the **User** and **Invoice** services are two separate entities, so you can't make them communicate through a simple function call, but you'll need to use the interface the services are providing: you need to make an API Call.

In this case, there are some additional considerations to keep in mind:

The network is now part of the game, with all its complexity and unreliability. We often make some assumptions that aren't true:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

- A service might be down for some reason, so it won't be able to fulfill the request
- In case you have multiple service replicas, how do you know what's the right service instance to contact?
- How can a service announce to the cluster that a new replica has been deployed?

The last two points actually point toward the *service discovery* topic

WHAT IS SERVICE DISCOVERY?

Service discovery is the process of automatically finding what instances of service fulfil a given query. You will invoke some service discovery process which will return a list of suitable servers.

In more distributed environments, the task starts to get more complex, and services that previously could blindly trust on their DNS lookups to find dependencies now have to deal with things like client-side load-balancing, multiple different environments (e.g. staging vs. production), geographically distributed servers, etc.

WHAT IS A SERVICE MESH SOLUTION?A service mesh solution is a dedicated infrastructure layer for handling service-to-service communication and is responsible for the reliable delivery of requests through the complex topology of services that comprise a cloud application.

A service mesh solution is typically comprised of: dynamic service discovery, load balancing, TLS termination, HTTP/2 & gRPC proxying, circuit breakers, health checks, staged rollouts with %-based traffic split, fault injection, and rich metrics.

There are a number of solutions out there on the web. Worth mentioning are Istio, Conduit and Linkerd

They all provide a set of common and unique features, but **they all share the basic implementation principle**.

The service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware. This proxy is then responsible for routing the request to the right service instance, making sure to retry the request for a number of times if it fails. It also ensures limited timeouts, enabling circuit breaker when it makes sense and so on.

WHAT ABOUT KUBERNETES?

How does this solution behave in a Kubernetes Cluster?

It turns out every node in a Kubernetes cluster runs a `kube-proxy`. `kube-proxy` is responsible for implementing a form of virtual IP for Services and, depending on the configuration mode, it can also listen to all the incoming and outcoming connections to each pod in your cluster.

Moreover, a Kubernetes service has as well an associated array of endpoints, representing all the pods associated with that service.

Istio, for example, is leveraging low level primitives to offer a service mesh solution in your cluster.

EXPRESS GATEWAY PLAYS NICELY WITH SERVICE MESH SOLUTIONS.

We've seen in previous articles how to put Express Gateway as a `DaemonSet` in Kubernetes in order to put an edge gate in the cluster.

Express Gateway can also be deployed per each service using a sidecar container (this pattern is usually called microgateway) so you can handle different concerns at HTTP level rather than TCP/UPD level.

What does this actually mean?

As you may already know, API gateways protect, enrich, and control access to API services. Developers can use API Gateways to architect applications in a way that provides clear separation between the business and security logic.

So, think of it this way: while API Gateways expose API/Edge services, a service mesh serves an inter-service communication infrastructure with no real business notion of your solution.

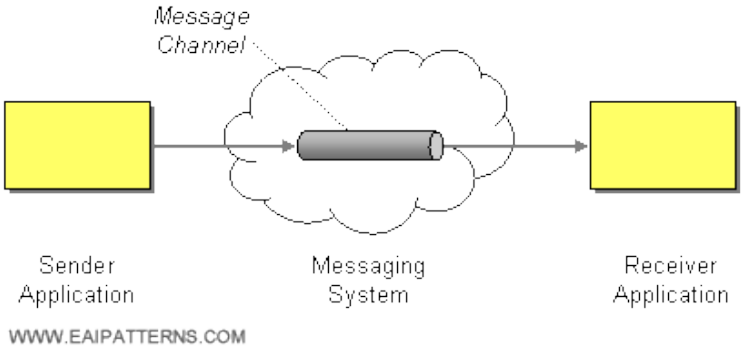
There are some overlapping capabilities. However; the real important thing to remember is that API Gateways, like Express Gateway, can serve to provide management of your APIs with inter-service communication of an API Gateway or it's also possible to use an API Gateway to can call downstream services via service mesh by sending application network functions to service mesh.

Enterprise Integration Patterns

Patterns are divided into seven sections:

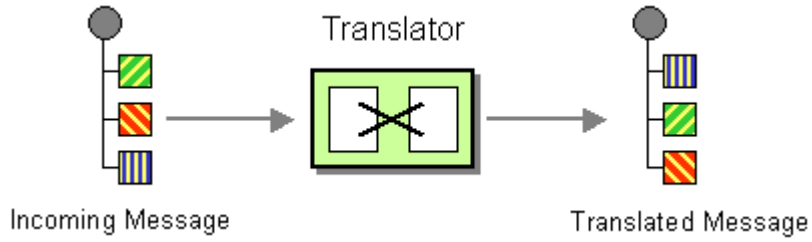
1. **Messaging Systems**,
2. **Messaging Channels**,
3. **Message Constructions**,
4. **Message Routing**,
5. **Message Transformation**,
6. **Messaging endpoints**,
7. **System management**.

Message Channel (from Messaging Systems)



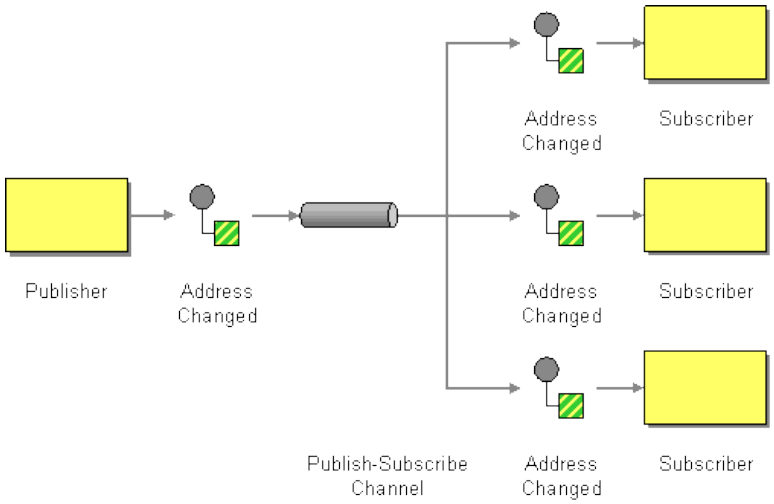
A message channel is a logical channel which is used to connect the applications. One application writes messages to the channel and the other one (or others) reads that message from the channel. Message queue and message topic are examples of message channels.

Message Translator (from Messaging Systems)



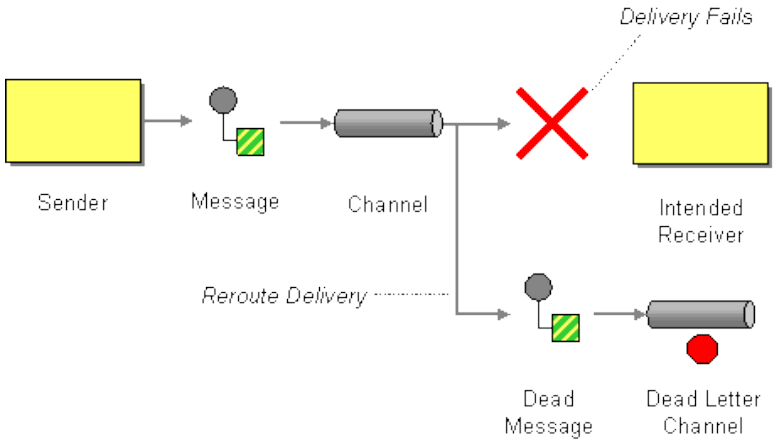
Message translator transforms messages in one format to another. For example one application sends a message in XML format, but the other accepts only JSON messages so one of the parties (or mediator) has to transform XML data to JSON. This is probably the most widely used integration pattern.

Publish-Subscribe Channel (from Messaging Channels)



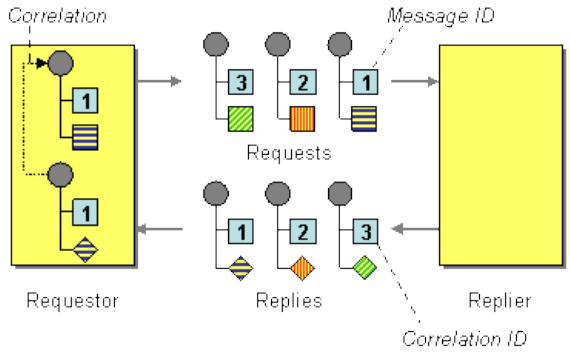
This type of channel broadcasts an event or notification to all subscribed receivers. This is in contrast with a point-to-point channel . Each subscriber receive the message once and next copy of this message is deleted from channel. The most common implementation of this patter is messaging topic.

Dead Letter Channel (from Messaging Channels)



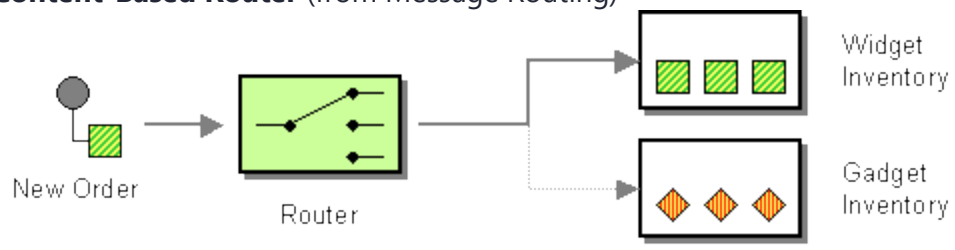
The Dead Letter Channel describe scenario, what to do if the messaging system determines that it cannot deliver a message to the specified recipient. This may be caused for example by connection problems or other exception like overflowed memory or disc space. Usually, before sending the message to the Dead Letter Channel, multiple attempts to redeliver message are taken.

Correlation Identifier (from Message Construction)



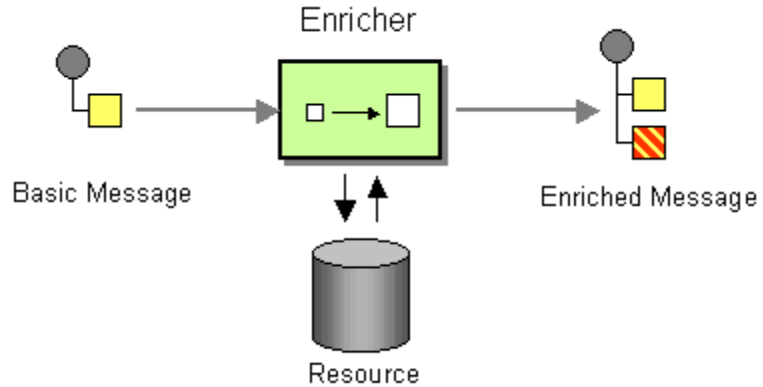
Correlation Identifier gives the possibility to match request and reply message when asynchronous messaging system is used. This is usually accomplished in the following way:
Producer: Generate unique correlation identifier.
Producer: Send message with attached generated correlation identifier.
Consumer: Process messages and send reply with attached correlation identifier given in request message.
Producer: Correlate request and reply message based on correlation identifier.

Content-Based Router (from Message Routing)



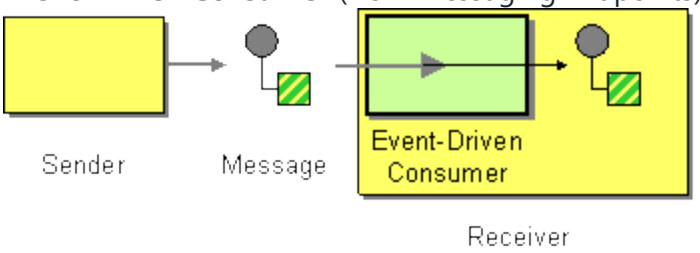
Content-Based Router examines message contents and route messages based on data contained in the message.

Content Enricher (from Message Transformation)



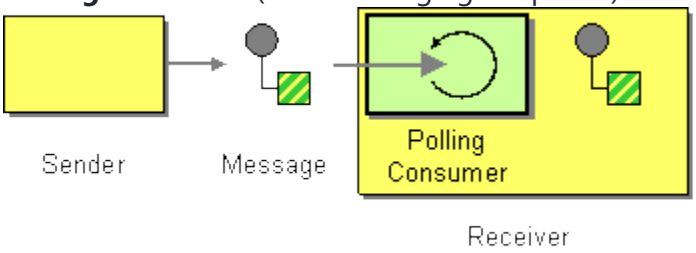
Content Enricher as the name suggests enrich message with missing information. Usually external data source like database or web service is used.

Event-Driven Consumer (from Messaging Endpoints)



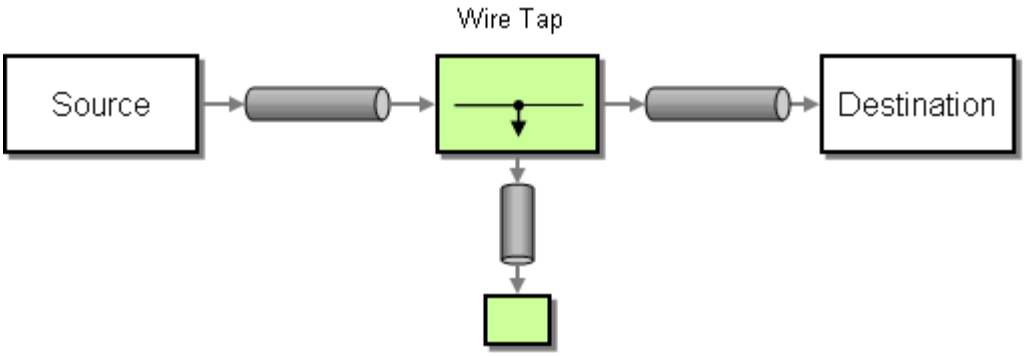
Event-Driver Consumer enables you to provide a action that is called automatically by the messaging channel or transport layer. It is asynchronous type of pattern because receiver does not have a running thread until a callback thread delivers a message.

Polling Consumer (from Messaging Endpoints)



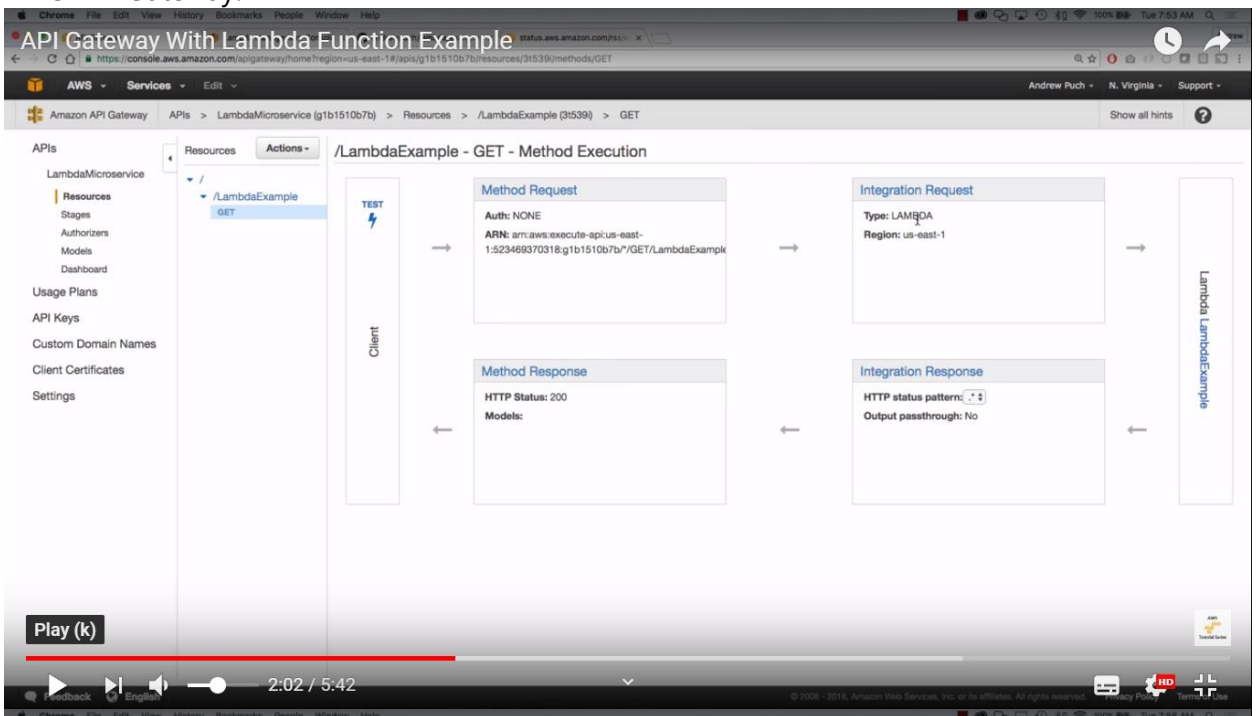
Polling Consumer is used when we want receiver to poll for a message, process it and next poll for another. What is very important is that this pattern is synchronous because it blocks thread until a message is received. This is in contrast with a event-driven consumer. An example of using this pattern is file polling.

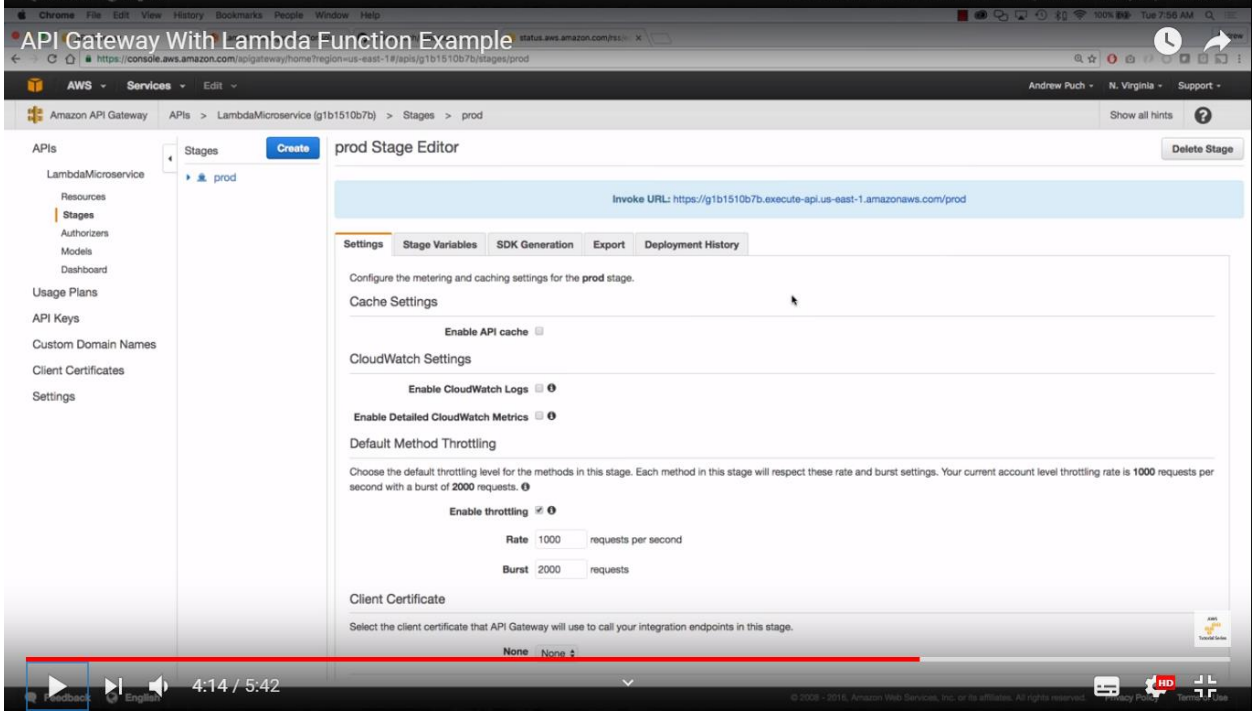
Wire Tap (from System Management)

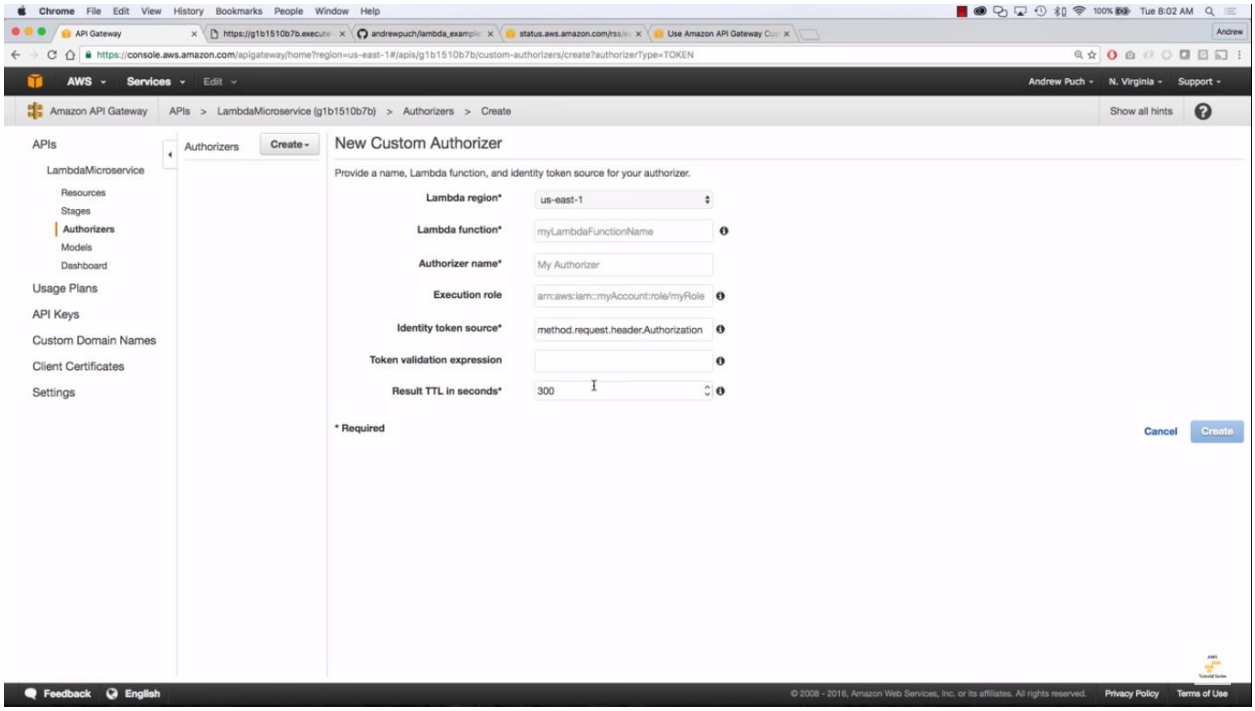


Wire Tap copy a message and route it to a separate channel, while the original message is forwarded to the destination channel. Usually Wire Tap is used to inspect message or for analysis purposes.

AWS API Gateway:






Chrome File Edit View History Bookmarks People Window Help
Tue 8:02 AM
API Gateway
https://g1b1510b7b.execute
andrewpuch/lambda_example
status.aws.amazon.com/rss/e
Use Amazon API Gateway Cus
Andrew
https://console.aws.amazon.com/apigateway/home?region=us-east-1#/apis/g1b1510b7b/custom-authorizers/create?authorizerType=TOKEN
AWS
Services
Edit
Andrew Puch
N. Virginia
Support
Amazon API Gateway
APIs > LambdaMicroservice (g1b1510b7b) > Authorizers > Create
Show all hints
APIs
LambdaMicroservice
Resources
Stages
Authorizers
Models
Dashboard
Usage Plans
API Keys
Custom Domain Names
Client Certificates
Settings
Authorizers
Create
New Custom Authorizer
Provide a name, Lambda function, and identity token source for your authorizer.
Lambda region*
us-east-1
Lambda function*
myLambdaFunctionName
Authorizer name*
My Authorizer
Execution role
arn:aws:iam::myAccount:role/myRole
Identity token source*
method.request.header.Authorization
Token validation expression
Result TTL in seconds*
300
* Required
Cancel
Create
Feedback
English
© 2008 - 2016, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Response(Status) Codes

1XX Codes: Informational Codes

These are the acknowledgement responses, used to pass on information. This status code is not used commonly.

2XX Codes: Success Codes

These are the success codes which means that the server has received the request from the client and processed it successfully.
For Example:

- **200 OK:** This informs the client about the successful response.
- **201 Created:** This should be returned for `POST` requests, stating that the resource is created successfully on the server. Like, when the client sends: `POST` **/myApp/students/** request and the student recourse is created successfully then the server should return `201 Created` as status code. The server can also return `200 OK`, but it's always good to be more precise, and send 201 Created in case of a POST request.
- **202 Accepted:** This informs the client that the request has been successfully received, but the processing is not yet finished.
- **204 No Content:** This informs the client that the request has been successfully processed, but no content will be returned.
- Check this for the complete list of 2XX status codes: 2XX Status Codes

---

3XX Codes:

These codes are generally used in case of URL Redirection.

4XX Codes:

This class of status codes are returned if the client's request has error. The request could be incorrect or the resource which the client is looking for doesn't exist.

- **400 Bad Request:** There is something wrong in the request from the client, hence the server cannot or will not process it.
- **401 Unauthorized:** The client needs to authorize themselves to make this request.
- **403 Forbidden:** This status code is used when the client request is correct but the server refuses to process the request. The client might not have required permissions.
- **404 Not Found:** The resource which the client is requesting for doesn't exist.

5XX Codes: Server error

The 5XX status codes are when server fails to process the request and cannot send the correct response. In this case the client doesn't have to change its request as the problem is with the server where the ~~REST~~ API is deployed.

- **500 Internal Server Error:** This is a generic status code returned when an unexpected condition is encountered on the server, while processing the request.
- **503 Service Unavailable:** When server is not available due to excessive load or may be down for maintainence, this status code is returned to the client.