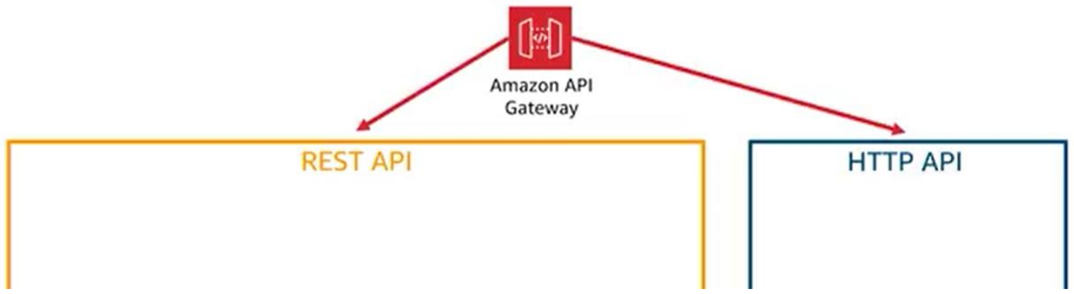
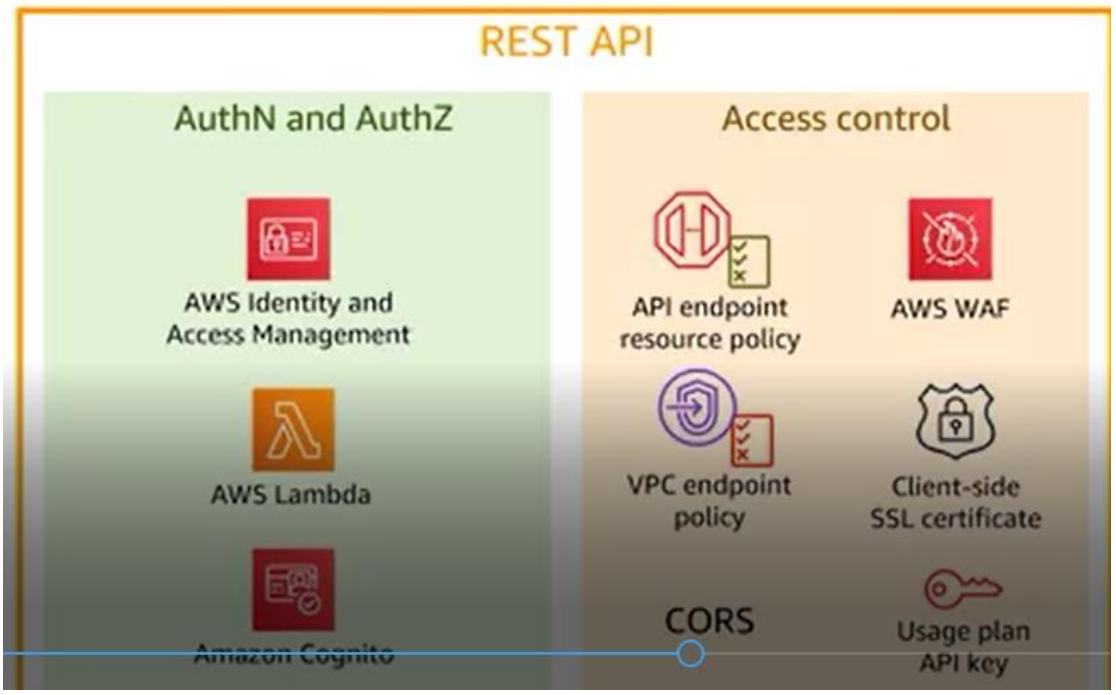
Authentication is the process of verifying who someone is, whereas authorization is the process of verifying what specific applications, files, and data a user has access to

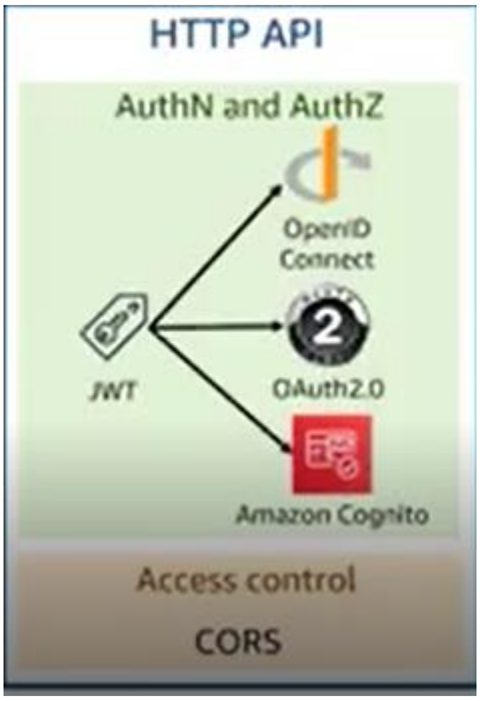


Identity and access management:
REST API: there are several methods:



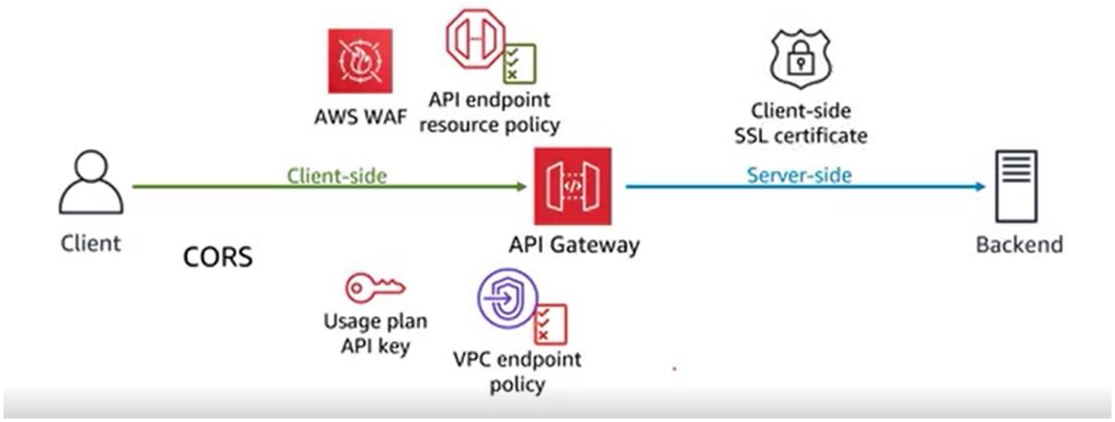
AWS WAF: Web application Firewall.

HTTP:



The only way is using JSON WEB Tokens (JWT) that are part of Open ID connect an OAuth2.0 protocols: this means we can integrate with third party identity providers.
Also because Amazon Cognito user pools uses the JWT as well, this can be used for authentication as well.
For authorization process, we can use the token return by OAuth2.0 identity provider.
Also HTTP supports the CORS

Different types of access control mechanisms in API Gateway:
Note: Access Control acts as addition to Auth-N and Auth-Z mechanisms
We can use the Access Control on top of AWS Identity Management, AWS Lambda, Amazon Cognito user pools.
And we don't have to choose between these two. We can even combine several access controls together.



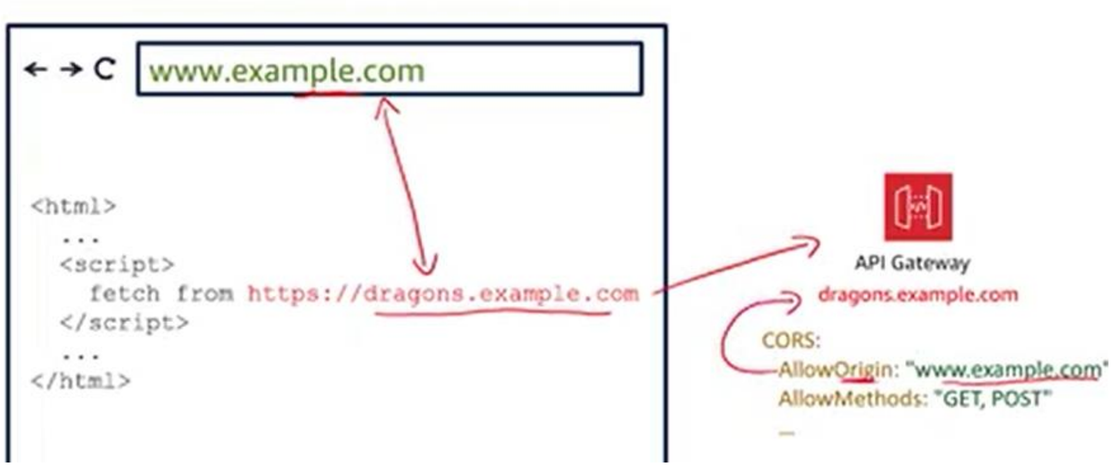
Note: all mechanism of access control except one are client side.

On server side, the only mechanism is client-side SSL certificate.
This is a mechanism to authenticate the API Gateway itself to backend, and backend will know that it is our API. So we will use this only if we need to authenticate the API Gateway to backend.

Client side Access Controls:

- CORS: Cross Origin Resource Sharing: is a browser security feature that restrict cross origin HTTP request that are running in the browser:

When the client is in 1:www.example.com and send the req to 2:httpss://dragons.example.com means that it executes the cross origin http req to dragons.example.com. Since the origin (URL) s are different between 1 and 2.
So by using and setting up CORS in API Gateway, we can ensure that the req are coming from origin 1, and query from 2 endpoint API.



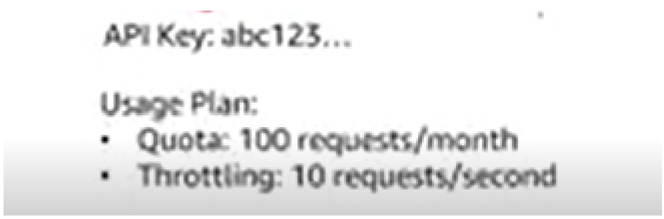
```
<html>
  ...
  <script>
    fetch from https://dragons.example.com
  </script>
  ...
</html>
```

In cases that the browser of client want to req from API Gateway, the CORS must be configured.

- AWS WAF: protect API from web exploits. We can use the managed rules to address who's like, last top security risk which are regularly updated as new issues emerged, it can also be configured to filter traffic based on rules that we define: i.e. we can filter any part of web requests such as IP addressed, http headers, http body. It is always very good idea to have web application firewall in front of our web applications.

- Usage Plan API Key: limit the access to authorize clients, to use it, we need to create API key in API Gateway then send it to client, this key is made of alpha-numerical characters. Client put this API key as a value of HTTP header X API key.

Using this only for authentication is not a good idea, so don't use it for authentication as it is not safe enough.
So it goes with usage plans we can add into the API key to allow customers to access selected APIs at agreed upon request rates and quota, like example:



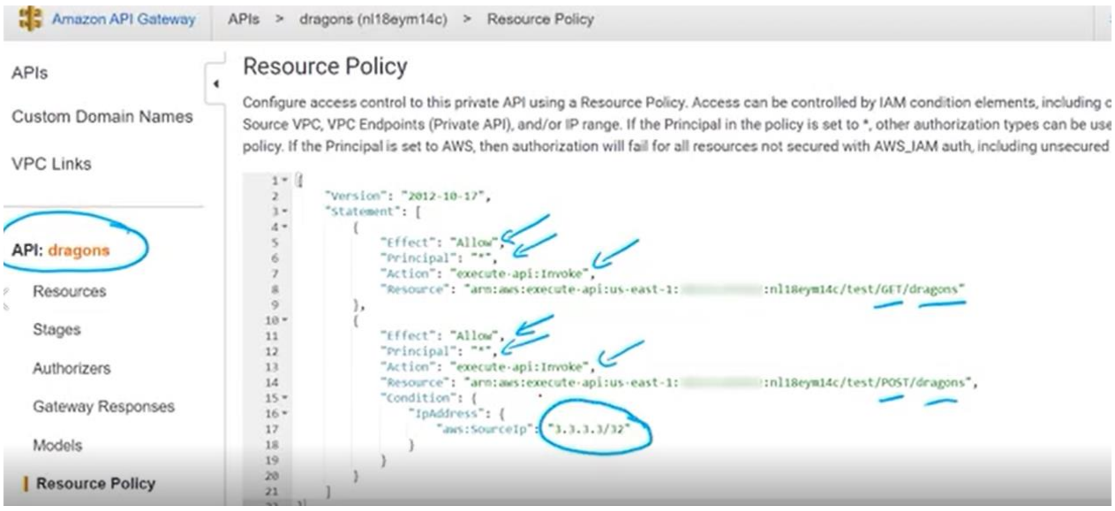
API Key: abc123...

Usage Plan:

- Quota: 100 requests/month
- Throttling: 10 requests/second

- Resource Policy: it allows you to create resource based policies to allow or deny access to your APIs and methods using im conditions elements, this include users from specific AWS account, specific source IP address range or specific virtual private cloud VPC endpoint

A few AWS services has this capability: simple storage service. If you have heard of bucket policies, this is very similar.
Example:



Resource Policy

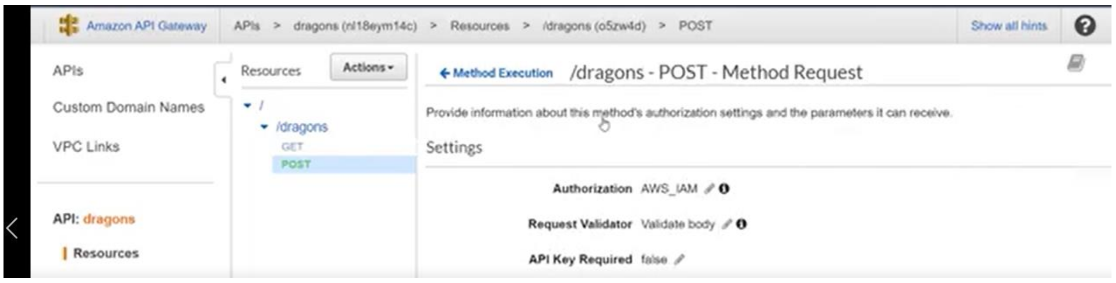
Configure access control to this private API using a Resource Policy. Access can be controlled by IAM condition elements, including c
Source VPC, VPC Endpoints (Private API), and/or IP range. If the Principal in the policy is set to *, other authorization types can be use
policy. If the Principal is set to AWS, then authorization will fail for all resources not secured with AWS_IAM auth, including unsecured

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:us-east-1:[redacted]:nl18eym14c/test/GET/dragons"
        },
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:us-east-1:[redacted]:nl18eym14c/test/POST/dragons",
            "Condition": {
                "IpAddress": {
                    "aws:SourceIp": "3.3.3.3/32"
                }
            }
        }
    ]
```

There is allow statement, not deny, but it is implicit deny, since we only allow some specific.
The resource policy access control is great for cross AWS account access and apply more restrictions on other authorizers, it is also very great to limit access to only range of IP addresses.

- VPC endpoint Policy:
  As we know, private API is only reachable from inside our VPC. The Policy is to restrict to the interface of VPC,
  To access this private API, for example in EC2 instance inside of EPC, u must use a VPC interface endpoint that are powered by AWS private link, which enables u to privately access AWS services by using private IP addresses, so keep the traffic entirely inside AWS and not exposing it to the internet. On those endpoints, we can add the policy to improve the security, for example allow some specific IP addresses to access VPC APIs.

Three Authorization and Authentication mechanisms: IAM, Lambda, Amazon Cognito.



/dragons - POST - Method Request

Provide information about this method's authorization settings and the parameters it can receive.

Settings

Authorization AWS_IAM

Request Validator Validate body

API Key Required false

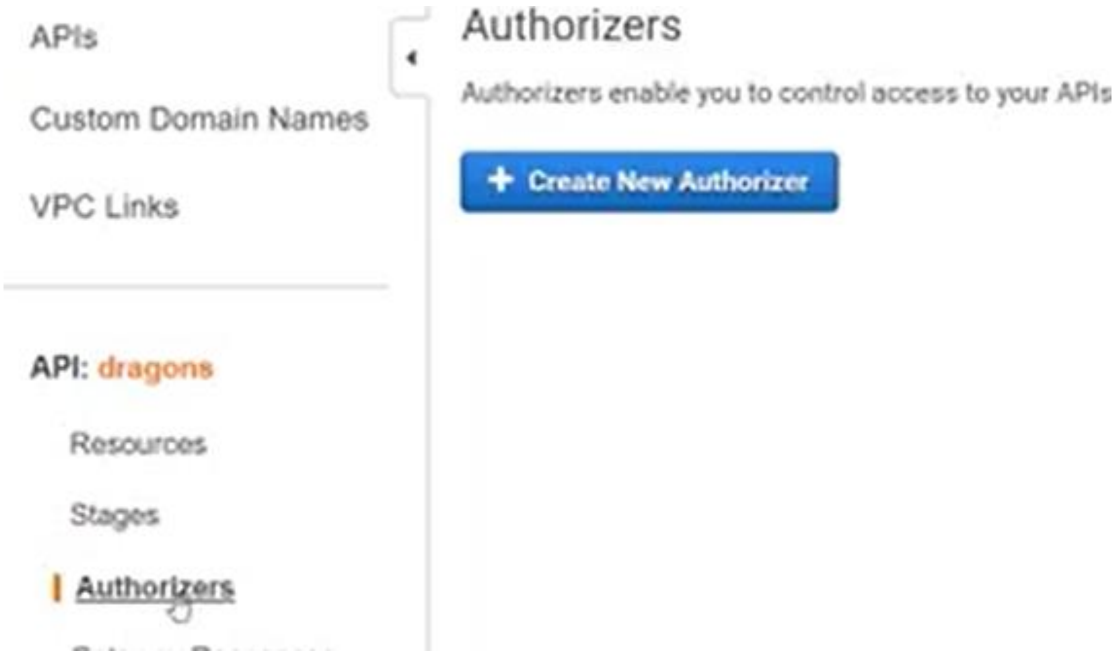
Note: we need to have Lambda or Amazon Congito user pool to have the authorization type of them.

Amazon IAM: this authorizer adds the same Auth-N and Auth-Z on top of API. This is done via access key, and secret key using the signature version 4 protocol. So access key and secret key creates the signature each time a req is sent to AWS. We don’t need to do it, since AWS SDK does this for us.

Although we can use the AWS SDK to manage the API like create it or update it, we can’t use the AWS SDK to invoke the API.

This type of AUTH_Z is for server-to-server communication, example: your code is running in container like Elastic container Service or an EC2 instance and it was making the call to your API, then u should AUTHN and AUTHZ it via IAM. The same way that our code make a call to service of AWS s3.

The best practice is to attach the IAM role to those services and use a temporary credentials generated and rotated via this IAM role for your code to get authenticated. The authorization will be made by IAM policy attached to that IAM role. In this policy, we can say credentials allow access one specific method and one specific stage. So IAM as an authorizer is made for server-to-server communication. For example your code running in AWS speaking to AWS Gateway and communicating with mirror server backend.



Authorizers enable you to control access to your APIs

Create new authorizer: so we can create Lambda or Cognito.

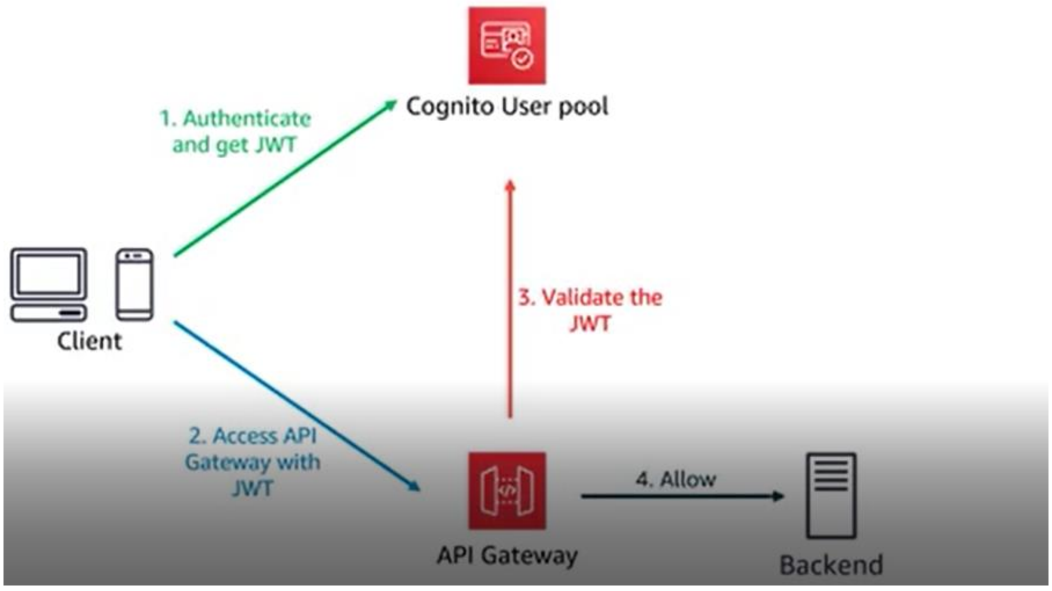
Cognito: to use this, we need to have Cognito user pool to have this type of authorizer
We can use this service to have users sign up or sign in. once the user signed in, the application will receive a JSON Web Token that can be sent to API gateway via the authorization header. We use this authentication for a web or mobile application to authenticate to API gateway.

Customization and flexibility: what if these authorizer missing the feature we want? Maybe we want to authenticate via customer authorizer:
So we use Lambda Authorizers.
It has greatest flexibility but with a price: we need to code it. Means piece of code is executed whenever request is made. So we can integrate with any authentication tools we want.
Example:
Client want to access the backend via API gateway. We also created our authorization code using AWS Lambda:

1. Client send request with parameters to API Gateway, parameters could be a token, username and password, , etc. we can have API gateway to use the regular expression to validate the format of those parameters.
2. After that the request is valid, these requests parameters send to my authorization code in Lambda. That code whatever it wants using those parameters, maybe it could call Okta or SAML identity provider or LDAP or active directory,
3. Then the code returns the principle and a policy, the principle is definition of this user, maybe it is username or user ID or anything that identifies this client uniquely. The policy is like IAM policy. It defines whatever the user is allowed to access, Note that the code not deny or allowing, it just defines the policy,
4. Then the policy goes to be evaluated and will be cached, the caching is useful, since the subsequent request doesn't need to go through the authorization code again,
5. Depending on the policy, for example the policy says the use is allowed, then it send to the backend, if the policy says it is denied, then it return 403 response code to client which means forbidden

Amazon Cognito user pool
The user pool: since Amazon Cognito Service is made up of two services: user pool and federated identities or identity pool. Although those two services can work together, but is important to distinguish them, and they also work individually.
Cognito user pool: is a user directory, which u can also referred to as an identity provider. It allows u to sign up users with email or phone number, sign in them, help them reset their password, and even ask them to do multi-factor authentication.
User Pool also uses an option for a hosted user interface. Instead of having to create your own web page for asking the users to put the username and password, AWS already created the web pages which u can customize and AWS will host it for u. once the user logged in, u can directly authenticate with certain services like API Gateway.

1. Client needs to authenticate to Amazon Cognito user pool, either by sending the credentials directly via its API or via the hosted authentication page.
2. Once authenticated, user pool send back an Auth to Jason Web Token(JWT), the client then can send a request to API to API Gateway through JWT token
3. API gateway validate that token with the user pool,
4. and if everything checks out, it will allow the requests to the backend



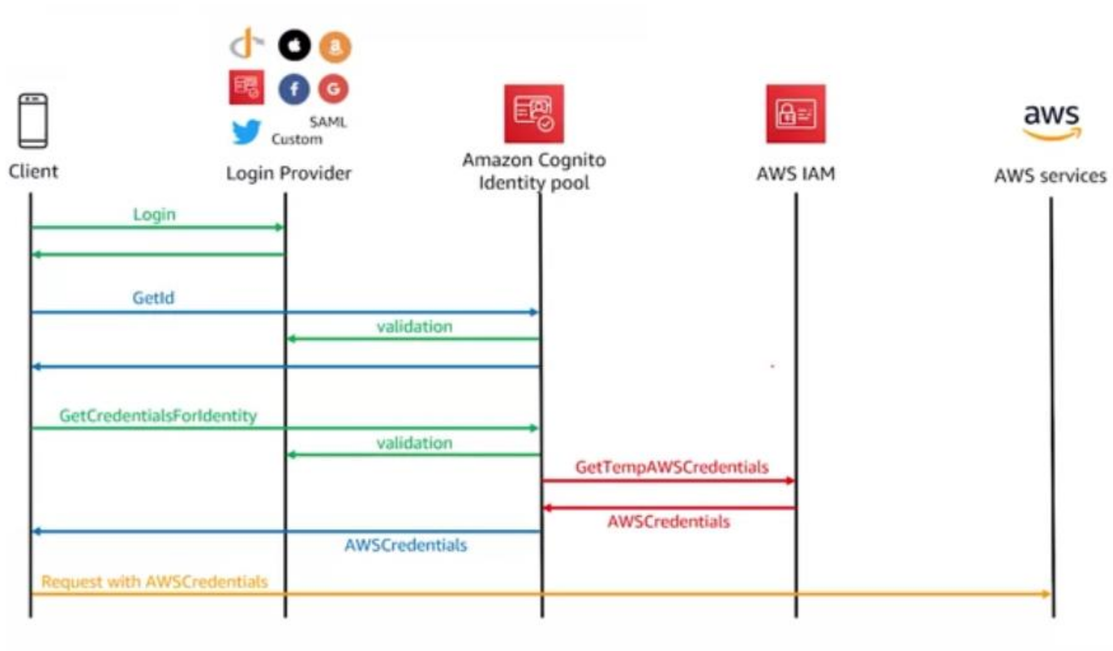
Another great feature of user pool is to add triggers to specify your code to launch for pretty much any piece of this workflow when it touches the user pool, for example we want to run the code once the user sign up, or u may want to execute the code every single time user authenticate towards the user pool,
Note: Identity federation is a system of trust between two parties for the purpose of authenticating users and conveying information needed to authorize their access to resources,
The last feature is the possibility to also do federation within user pools, instead of authenticating the user on user pool itself and host username and password, u can have them authenticate via a third party. This way user no need to create them to remember another set of credentials:

1. client authenticate via Open ID Connect, SAML or some social sign-in providers like sign in with Amazon, Apple, Google, Facebook, after the user sign in, the client receive the token. One of the greatest features of the hosted authentication UI, it can integrate with the social sign in providers,
2. client sent a request to Cognito user pool to exchange the token and receive from one of those third party and get JWT token in return,
3. before providing that JWT token, Cognito will validate that with the provider,
4. Then it will create a copy of information it received (user info), It will return the JWT token to the client,

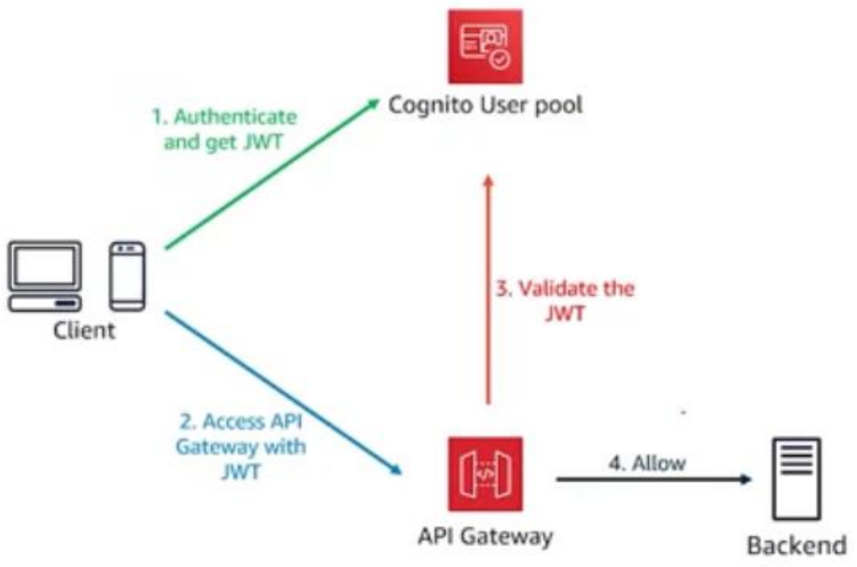
5. Client use that JWT to get authenticated and authorized to API Gateway,
6. which again validate the JWT token with user pool,
7. finally allow or deny access to backend

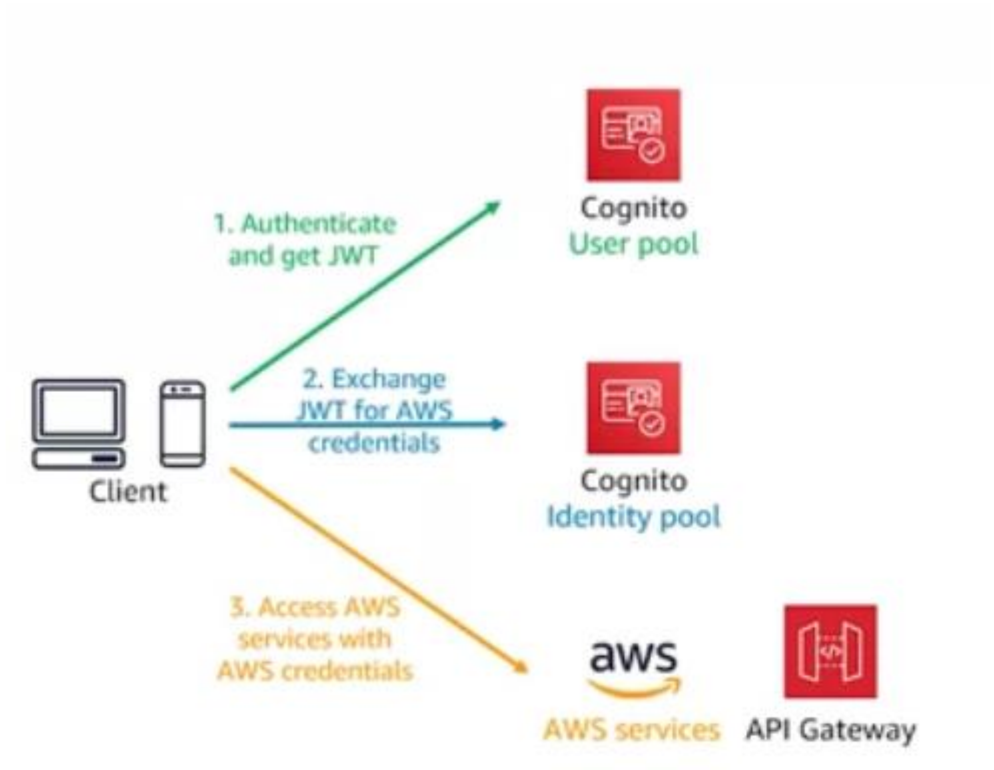
Why not directly integrate my application (backend) with social providers, why we use the Cognito user pool?
One of the reasons is for authorization, we can use third party login to access the backend, what we cannot give certain users different permissions, so for this we use the user pool, in above, in step 4, we can assign some permissions or authorization scopes whining Cognito user pool.
Federated identities (**identity pool**): This is a service that's used to assign IAM roles to users who authenticate through a separate identity provider. It doesn't contain your users and all their information like Cognito user pool, so this is major difference, also there is build-in rule engine that allows u to select a specific IAM role based on data provided by the identity provider. This means that u could provide one role with more permissions for a certain group of users and less permissions for another by using different roles,

1. Client log in to a login provider (third party, custom provider, Cognito user pool). So if we choose the Cognito user pool, and since we have identity pool, it means they can work together.
2. The response will be sent back to client,
3. Then the client send the API call get ID to identity pool using the token it received from login provider,
4. That token will be validated with the login provider and identity pool will let the client know about it,
5. Next client use the API call get credentials for identity request for specifying what IAM role it will like to become, this will be validated against the login provider,
6. and if it is allowed, Cognito identity pool will request AWS request credentials from AWS IAM, which then will be returned to Cognito identity pool and client,
7. finally client can use those credentials to access any AWS services,



Note: we have mentioned that we can authorize Cognito user pool to API gateway via JWT, now with identity pool, we can actually authenticate with one of these login providers which includes Cognito user pool to also be able to authenticate with any services, which includes API Gateway, > here we see two different ways to authenticate API Gateway, by using the Cognito user pool we can authenticate only the API Gateway, but with identity pool we can authorize to any AWS services.. And also pay attention to select this type of authentication within API gateway, we select the AWS IAM.




1. Authenticate and get JWT
Cognito
User pool
2. Exchange JWT for AWS credentials
Cognito
Identity pool
Client
3. Access AWS services with AWS credentials
aws
AWS services
API Gateway

The last feature: unauthenticated identity:
U can give a default IAM role for anyone that doesn't have credentials to authenticate. For example provide very small amount of access for users not authenticated.