

API Gateway authorization example

aws api gateway custom authorization example from scratch step by step in chat

Got it 👍 — let's build an **AWS API Gateway Custom Authorizer** (a Lambda-based authorizer) step by step from scratch. I'll show you the **Token-based Custom Authorizer** example, since it's the most common starting point.

♦ Step 1: Setup API Gateway

1. Log in to **AWS Management Console**.
2. Go to **API Gateway** → Click **Create API**.
3. Choose **HTTP API** or **REST API** (I'll use REST API here since it's easier for beginners).
4. Give it a name, e.g., `MyCustomAuthAPI`.

♦ Step 2: Create a Lambda Authorizer Function

We'll create a Lambda function that validates a token (e.g., `"allow-token"` for demo).

Example in Python

```
python

import json

def lambda_handler(event, context):
    token = event['authorizationToken'] # incoming token from header
    method_arn = event['methodArn']    # API method being invoked

    # Example token check (in real apps, verify JWT or DB lookup)
    if token == "allow-token":
        effect = "Allow"
    else:
        effect = "Deny"

    return generate_policy("user", effect, method_arn)
```

```
def generate_policy(principal_id, effect, resource):
    auth_response = {}
    auth_response['principalId'] = principal_id

    if effect and resource:
        policy_document = {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "execute-api:Invoke",
                    "Effect": effect,
                    "Resource": resource
                }
            ]
        }
        auth_response['policyDocument'] = policy_document

    return auth_response
```

✓ What this does:

- Reads the **Authorization header token**.
- If it matches "allow-token" , grants access.
- Else, denies access.

◆ Step 3: Deploy Lambda Authorizer

1. Go to **AWS Lambda** → Create function.
2. Choose **Author from scratch** → Name: `MyAuthLambda` .
3. Runtime: `Python 3.x` (or Node.js if you prefer).
4. Paste the code above.
5. Deploy.

◆ Step 4: Attach Authorizer to API Gateway

1. In **API Gateway**, go to your API.
2. Under **Authorizers**, click **Create New Authorizer**.
 - Name: `MyCustomAuthorizer`

- Type: **Lambda**
- Lambda Function: `MyAuthLambda`
- Token Source: `Authorization` (this means API Gateway looks for an `Authorization` header).

3. Save.

◆ Step 5: Secure an Endpoint with Custom Authorizer

1. In your API, create a **Resource** → e.g., `/hello`.
 2. Add a **GET Method**.
 3. Under **Authorization**, choose `MyCustomAuthorizer`.
 4. Attach a mock or Lambda integration (e.g., a test Lambda that just returns `"Hello World"`).
 5. Deploy API to a stage (e.g., `dev`).
-

◆ Step 6: Test API with Custom Authorization

Use **curl** or **Postman**:

✓ Allowed request:

```
bash

curl -H "Authorization: allow-token" https://<api-id>.execute-api.
<region>.amazonaws.com/dev/hello
```

✗ Denied request:

```
bash

curl -H "Authorization: wrong-token" https://<api-id>.execute-api.
<region>.amazonaws.com/dev/hello
```

◆ Step 7: Extend for Real Use (JWT Example)

- Instead of checking `"allow-token"`, you can:
 - Verify a **JWT token** with a secret (using `PyJWT` or `jsonwebtoken`).

- Call a database to validate a user session.
- Integrate with Cognito or Auth0.

👉 That's the **basic working example**.

Would you like me to also show you the **JWT-based Authorizer** version (with real verification), or keep it as this simple token-based example?

yes

Perfect 👍 Let's extend the **Custom Authorizer** to a real-world **JWT validation** example (instead of a static `"allow-token"`).

We'll use **PyJWT** in Python for verification.

◆ Step 1: Install Dependencies

In your Lambda, you'll need `PyJWT` .

Since Lambda doesn't have it by default:

- Create a project folder locally:

```
bash

mkdir jwt_auth && cd jwt_auth
pip install pyjwt -t .
zip -r jwt_auth.zip .
```

- Upload `jwt_auth.zip` to Lambda.

◆ Step 2: JWT Authorizer Lambda (Python)

```
python

import json
import jwt # PyJWT
from jwt import ExpiredSignatureError, InvalidTokenError

SECRET_KEY = "mysecretkey" # In real apps, store in AWS Secrets Manager
```

```

def lambda_handler(event, context):
    token = event.get("authorizationToken")
    method_arn = event["methodArn"]

    try:
        # Decode and validate JWT
        payload = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])

        # If valid, allow access
        return generate_policy(payload["sub"], "Allow", method_arn)

    except ExpiredSignatureError:
        return generate_policy("user", "Deny", method_arn)
    except InvalidTokenError:
        return generate_policy("user", "Deny", method_arn)

def generate_policy(principal_id, effect, resource):
    return {
        "principalId": principal_id,
        "policyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "execute-api:Invoke",
                    "Effect": effect,
                    "Resource": resource
                }
            ]
        }
    }

```

◆ Step 3: Generate JWT for Testing

On your local machine:

```

python

import jwt, datetime

```

```
SECRET_KEY = "mysecretkey"

token = jwt.encode(
    {"sub": "user123", "exp": datetime.datetime.utcnow() + datetime.timedelta(minutes=5)},
    SECRET_KEY,
    algorithm="HS256"
)

print(token)
```

This gives you a JWT like:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

◆ Step 4: Test API

✓ Valid token:

```
bash

curl -H "Authorization: <paste-your-jwt>" https://<api-id>.execute-api.
<region>.amazonaws.com/dev/hello
```

✗ Invalid/expired token:

```
bash

curl -H "Authorization: wrong-token" https://<api-id>.execute-api.
<region>.amazonaws.com/dev/hello
```

◆ Step 5: Best Practices

- Store `SECRET_KEY` in **AWS Secrets Manager** or **SSM Parameter Store** (not hardcoded).
 - Use **public/private keys** (RS256) for external IdPs like Auth0, Google, or Cognito.
 - Set **TTL (caching)** in Authorizer settings to avoid decoding JWT on every request.
-

👉 Now you've got:

1. **Simple static token-based Authorizer** ✓
2. **JWT-based Custom Authorizer with PyJWT** ✓

Do you want me to also show the **RS256 public key verification** example (like you'd use with Auth0 or Cognito), or keep it at HS256 for now?

yes

Awesome 🚀 Let's go one step further and do a **JWT Custom Authorizer with RS256 public key verification** (real-world case, like **Cognito, Auth0, Okta, Google Identity Platform**).

◆ Step 1: Why RS256?

- **HS256** → uses a **shared secret key** (not secure across multiple services).
- **RS256** → uses a **private key** to sign JWT and a **public key (JWKS)** to verify.
✓ Safer for distributed systems.

Most IdPs (Auth0, Cognito, Google) publish their **JWKS endpoint** that exposes public keys.

◆ Step 2: Example JWT Authorizer Lambda (RS256, Python)

We'll fetch the JWKS (JSON Web Key Set) and verify the JWT.

```
python

import json
import jwt
import requests
from jwt import ExpiredSignatureError, InvalidTokenError
from jwt.algorithms import RSAAlgorithm

# Replace with your provider's JWKS URL (example: Auth0 or Cognito)
JWKS_URL = "https://YOUR_DOMAIN/.well-known/jwks.json"
```

Cache JWKS so we don't fetch on every request

jwks_cache = {}

def lambda_handler(event, context):

token = event.get("authorizationToken")

method_arn = event["methodArn"]

try:

Decode header to get key id (kid)

unverified_header = jwt.get_unverified_header(token)

kid = unverified_header["kid"]

Load JWKS (fetch once and cache)

if not jwks_cache:

response = requests.get(JWKS_URL)

keys = response.json()["keys"]

for key in keys:

jwks_cache[key["kid"]] = key

if kid not in jwks_cache:

raise InvalidTokenError("Unknown kid")

public_key = RSAAAlgorithm.from_jwk(json.dumps(jwks_cache[kid]))

Verify JWT with public key

payload = jwt.decode(token, public_key, algorithms=["RS256"], audience="your-audience")

return generate_policy(payload["sub"], "Allow", method_arn)

except ExpiredSignatureError:

return generate_policy("user", "Deny", method_arn)

except InvalidTokenError:

return generate_policy("user", "Deny", method_arn)

def generate_policy(principal_id, effect, resource):

return {

"principalId": principal_id,

"policyDocument": {

"Version": "2012-10-17",

"Statement": [


```
{
  "Action": "execute-api:Invoke",
  "Effect": effect,
  "Resource": resource
}
]
```

◆ Step 3: Configure API Gateway

1. Attach this Lambda as your **Custom Authorizer**.
2. Set `Authorization` header as the token source.
3. Apply to your `/hello` endpoint.

◆ Step 4: Test with Real JWT

- If using **Auth0**, get a test token:

```
bash

curl --request POST \
  --url https://YOUR_DOMAIN/oauth/token \
  --header 'content-type: application/json' \
  --data
  '{"client_id":"YOUR_CLIENT_ID","client_secret":"YOUR_CLIENT_SECRET","audience":"your-
  audience","grant_type":"client_credentials"}'
```



- Copy `access_token` and test:

```
bash

curl -H "Authorization: Bearer <your-jwt>" https://<api-id>.execute-api.
<region>.amazonaws.com/dev/hello
```

◆ Step 5: Best Practices

-  Cache JWKS keys (to avoid fetching on every request).

-  Use **audience** and **issuer** validation (`aud` , `iss` claims).
 -  Enable **Authorizer caching in API Gateway** (e.g., 5 minutes) for performance.
-

👉 Now you have all **three levels** of API Gateway Custom Authorizers:

1. **Simple token-based** (demo/prototyping).
 2. **JWT HS256** (secret key).
 3. **JWT RS256** (public key from JWKS, used in real-world with Auth0/Cognito/Google).
-

Do you want me to also create a **step-by-step example with AWS Cognito JWKS URL** (most common in AWS setups), so you can try it directly in your AWS account?

yes

Perfect 👍 Let's now do a **real AWS Cognito + API Gateway Custom Authorizer example** step by step.

This is the most common setup in AWS projects.

◆ Step 1: Create a Cognito User Pool

1. Go to **Cognito** → **Create User Pool**.
 2. Choose **User pool only**.
 3. Give it a name (e.g., `MyUserPool1`).
 4. Enable **Username or Email** login.
 5. Leave defaults, click **Create pool**.
 6. Copy the **Pool ID** and **Pool ARN**.
-

◆ Step 2: Create an App Client

1. Inside the User Pool → **App clients** → **Create app client**.
2. Give it a name (e.g., `myappclient`).
3. Disable "Generate client secret" (API Gateway can't send secrets).
4. Save, copy the **Client ID**.

◆ Step 3: Find JWKS URL

Cognito publishes its JWKS at:

```
bash
```

```
https://cognito-idp.<region>.amazonaws.com/<userpool_id>/.well-known/jwks.json
```

Example:

```
bash
```

```
https://cognito-idp.ap-south-1.amazonaws.com/ap-south-1_123456/.well-known/jwks.json
```

◆ Step 4: Create a Custom Authorizer Lambda

We'll verify JWT tokens from Cognito using that JWKS.

```
python
```

```
import json
import jwt
import requests
from jwt import ExpiredSignatureError, InvalidTokenError
from jwt.algorithms import RSAAlgorithm

# Replace with your Cognito User Pool region & ID
COGNITO_POOL_ID = "ap-south-1_123456"
COGNITO_REGION = "ap-south-1"
APP_CLIENT_ID = "xxxxxxxxxxxxxx" # Your App Client ID (audience check)

JWKS_URL = f"https://cognito-idp.
{COGNITO_REGION}.amazonaws.com/{COGNITO_POOL_ID}/.well-known/jwks.json"
jwks_cache = {}

def lambda_handler(event, context):
    token = event.get("authorizationToken")
    method_arn = event["methodArn"]

    try:
        # Decode header to get key id
```

```

unverified_header = jwt.get_unverified_header(token)
kid = unverified_header["kid"]

# Load JWKS (fetch once and cache)
if not jwks_cache:
    response = requests.get(JWKS_URL)
    keys = response.json()["keys"]
    for key in keys:
        jwks_cache[key["kid"]] = key

if kid not in jwks_cache:
    raise InvalidTokenError("Unknown kid")

public_key = RSAAlgorithm.from_jwk(json.dumps(jwks_cache[kid]))

# Verify JWT (id_token or access_token from Cognito)
payload = jwt.decode(
    token,
    public_key,
    algorithms=["RS256"],
    audience=APP_CLIENT_ID, # validates "aud" claim
    issuer=f"https://cognito-idp.
{COGNITO_REGION}.amazonaws.com/{COGNITO_POOL_ID}"
)

return generate_policy(payload["sub"], "Allow", method_arn)

except ExpiredSignatureError:
    return generate_policy("user", "Deny", method_arn)
except InvalidTokenError:
    return generate_policy("user", "Deny", method_arn)

def generate_policy(principal_id, effect, resource):
    return {
        "principalId": principal_id,
        "policyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "execute-api:Invoke",
                    "Effect": effect,

```

```
    "Resource": resource
  }
]
}
}
```

◆ Step 5: Deploy Lambda

1. Create a Lambda function (`CognitoAuthLambda`) in AWS Lambda.
2. Upload code (with dependencies: `pip install pyjwt requests -t . && zip -r lambda.zip .`).
3. Deploy.

◆ Step 6: Configure API Gateway

1. In **API Gateway**, go to your API.
2. Add **Authorizer**:
 - Type: **Lambda**
 - Lambda: `CognitoAuthLambda`
 - Token Source: `Authorization`
3. Apply Authorizer to your endpoint (e.g., `/hello` → GET).

◆ Step 7: Get a Cognito JWT Token

Use Cognito's Hosted UI or initiate Auth with AWS CLI:

```
bash

aws cognito-idp initiate-auth \
  --auth-flow USER_PASSWORD_AUTH \
  --client-id <APP_CLIENT_ID> \
  --auth-parameters USERNAME=<username>,PASSWORD=<password> \
  --region ap-south-1
```

It will return:

```
json
```

```
{
  "AuthenticationResult": {
    "AccessToken": "eyJraWQiOiJrZXkxIiwiaWQiOiUIMyNTYifQ...",
    "IdToken": "...",
    "RefreshToken": "...",
    "ExpiresIn": 3600
  }
}
```

◆ Step 8: Test API

✓ Valid Cognito token:

```
bash
```

```
curl -H "Authorization: eyJraWQiOiJrZXkxIiwiaWQiOiUIMyNTYifQ..." https://<api-id>.execute-api.ap-south-1.amazonaws.com/dev/hello
```

✗ Invalid token:

```
bash
```

```
curl -H "Authorization: wrong-token" https://<api-id>.execute-api.ap-south-1.amazonaws.com/dev/hello
```

◆ Best Practices

- Enable **caching** in API Gateway Authorizer for 5–10 minutes.
- Use **AccessToken** for API calls (not `IdToken`).
- Validate both **audience (App Client ID)** and **issuer**.
- Store Cognito pool info in **Lambda environment variables**.

✓ Now you have a **real Cognito-backed API Gateway Custom Authorizer**.

This is production-grade and works for securing APIs with Cognito users.

Do you want me to also give you the **Node.js version** of this Cognito Authorizer Lambda (instead of Python), in case you prefer JS/TS for AWS projects?