BLOG 1

构建基于Auth0的统一身份管控平台实现AWS控制台的登录与授权

-- 第一篇 RBAC&ABAC

越来越多的企业在迁移上云旅程中都会面临多云的选择。不管是出于主动原因希望融合多云的服务优势，还是出于被动原因需要适应公司的业务并购，抑或是因为上云过程中需要实现混合云架构的打通，构建多云的统一身份权限管控平台往往是这些企业需要解决的问题。

构建这一平台，不仅要实现单点登录，还要考虑统一身份管控、统一权限管控和云上用户行为审计；并且企业内人员的多组织管理模型通常与云上基于角色的IAM管理模型存在较大差异，这进一步增加了构建跨云身份管理平台的难度。

本博客从企业客户的实际需求和技术难点出发，分别阐述了使用

- RBAC -- 基于角色的权限管控
- ABAC -- 基于属性的权限管控
- 动态策略生成

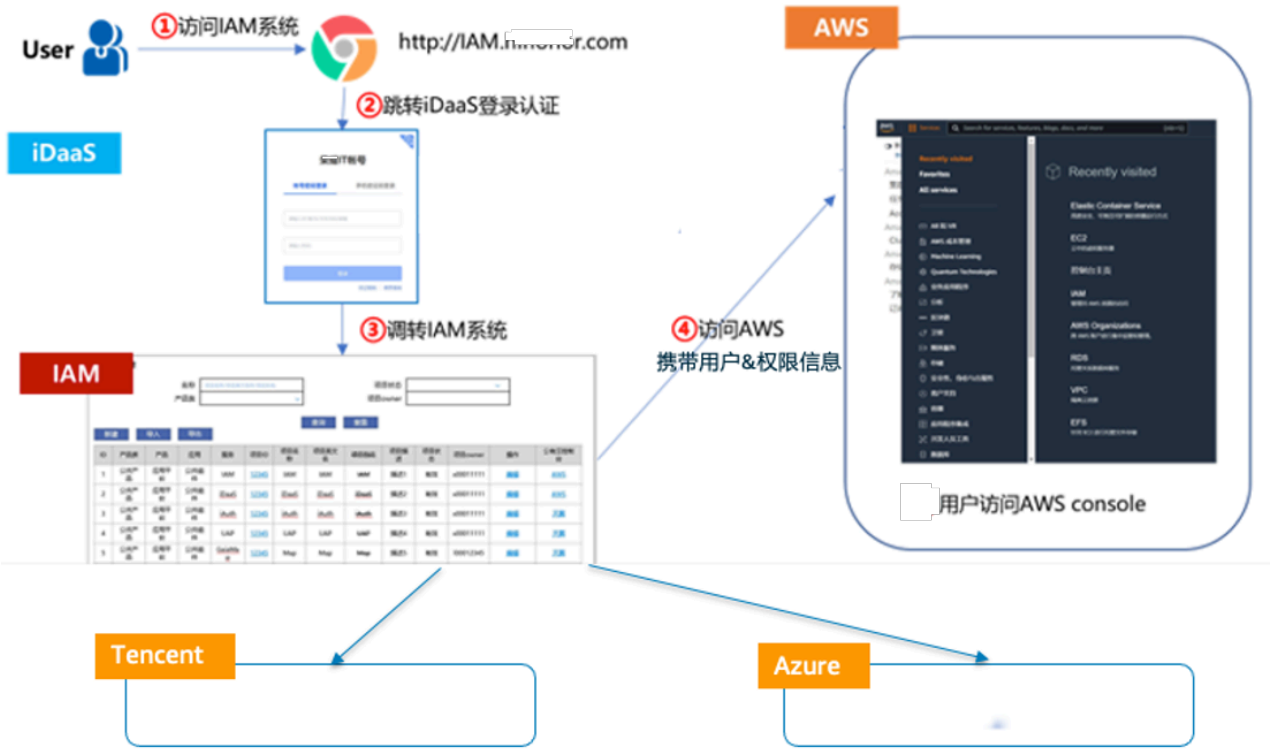
这三种思路实现企业统一身份管控平台，并与公有云控制台(Web console)实现身份和权限的打通；以及三种思路各自的适用场景；并在此基础上给出了基于Auth0构建该平台，完成与AWS控制台集成，实现统一登录与授权的具体实现代码样例。

受篇幅所限，我们把博客分成两部分，本篇讲述RBAC和ABAC的实现方法；第二篇讲述动态策略生成的实现方法。

1.统一身份管控平台概述

1.1 构建需求

下图展示了一个实际案例中企业对统一身份管控平台的需求：



1. 企业构建统一的身份、权限管控平台，并提供唯一入口对外访问
2. 用户访问统一身份管控平台后，自动跳转至企业iDaaS系统，完成身份认证
3. 认证成功后，进入企业的权限管理系统，获取相应的权限信息
4. 用户选择需要登录的公有云链接，携带用户及权限信息，实现对公有云控制台的无缝跳转和权限控制。

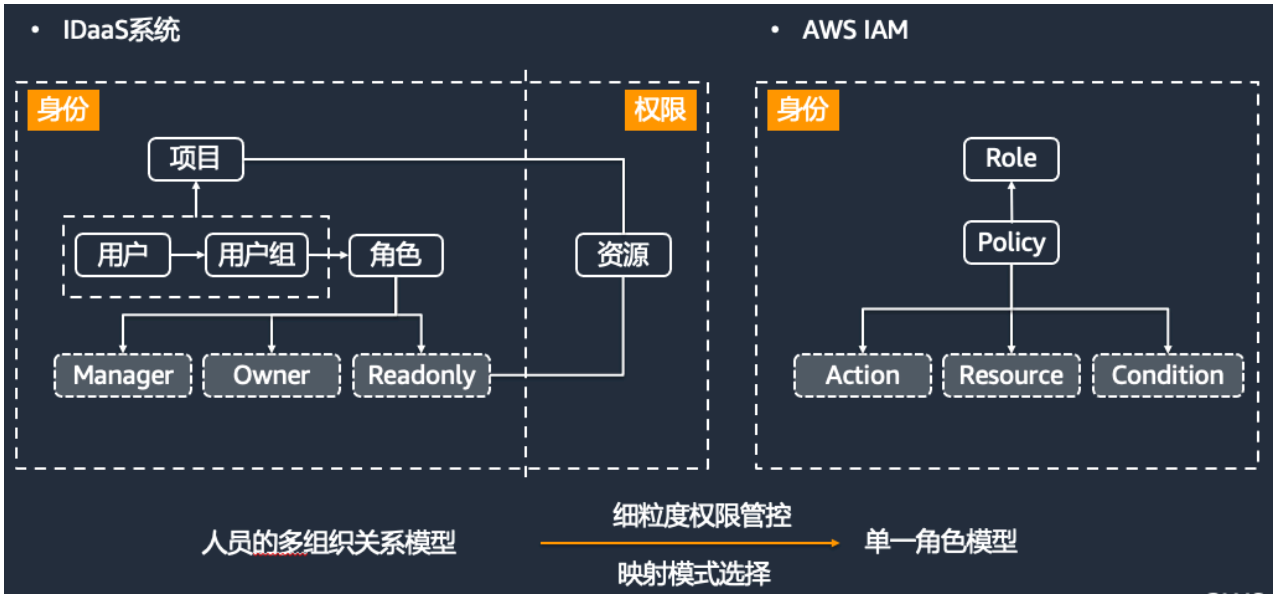
1.2 技术挑战

企业的iDaaS系统通常会采用人员的多组织关系模型。如下图左半部分所述的企业实际场景：

- 企业员工可以同时归属到多个项目，并在项目中担任不同角色
- 同一个员工在不同项目中可以担任不同角色。如在Project1中担任Manager，负责iDaaS系统中用户创建和权限分配；在Project2中担任Owner，负责公有云资源的创建和管理；在Project3中担任Readonly角色，对公有云资源具有只读权限。

下图右半部分以AWS IAM为例简要展示了公有云的IAM权限管理模型：IAM通过policy控制对云资源可以实施的动作，并通过条件对资源、行为主体、请求类型等进行细粒度限制；然后将policy挂接到Role上实现权限与行为主体的映射。

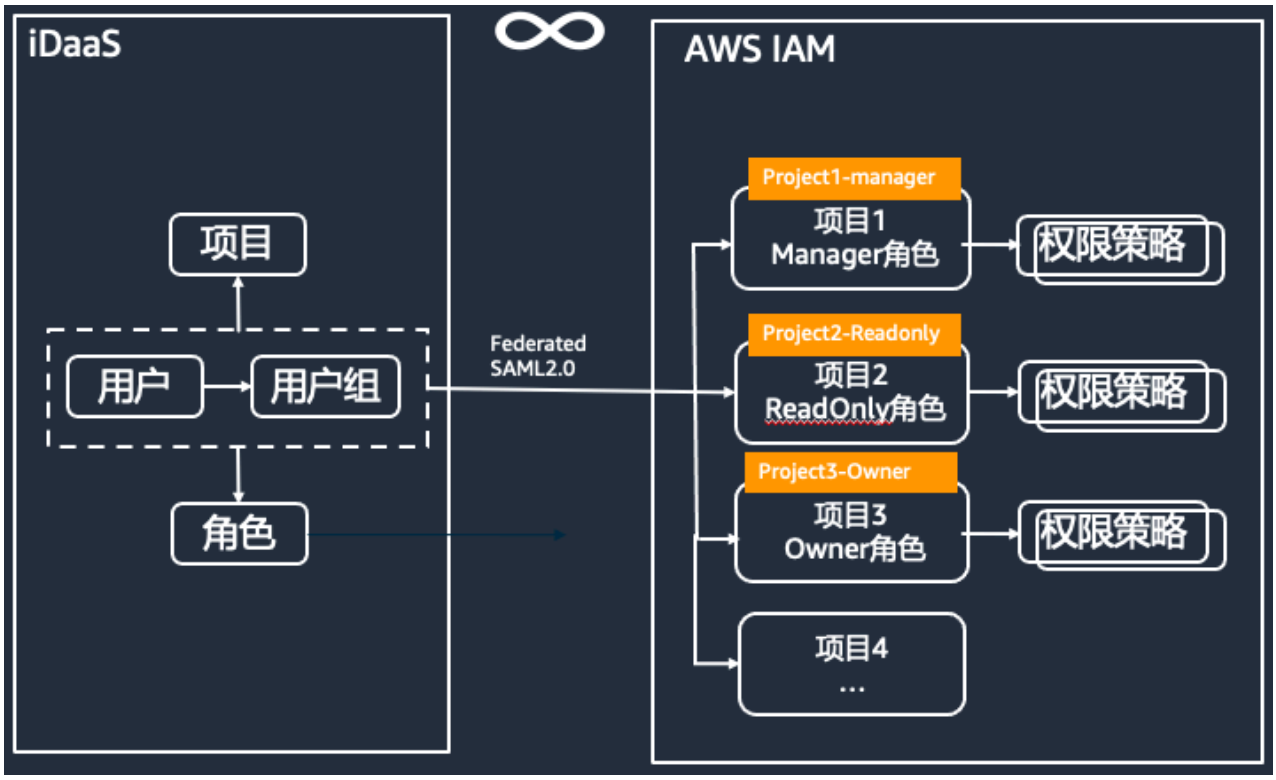
可以看到，公有云的IAM管理模型虽然具有用户、组和角色的划分，但账号内通常不会支持多级人员管理；而且SSO一般是通过IDP的用户属性到公有云IAM角色的映射实现权限打通，如何使用扁平的角色管理模型实现多组织层级的资源权限管控是当前面临的主要技术难点。



2. 思路一：基于角色的权限管理模型RBAC

2.1 方案概述

基于RBAC的方案是对企业iDaaS中的每一个项目x角色的组合在云中都对应一个指定的role。因此云中IAM角色的数量将与企业iDaaS中项目x角色的数量相等。如下图，使用RBAC，就是对企业中的每个项目都在AWS IAM中创建三个角色（Manager，Owner，Readonly），并为每个角色维护单独的权限策略。该方案的特点是通过静态角色完成设置，实现方法简单直接，非常适合项目和角色数量有限的场景；但随着项目和角色数量的增加，维护众多的的权限策略将逐渐成为噩梦。并且每账户的IAM角色数量通常具有上限，例如AWS的目前上限为每账户允许创建最多5000个角色.

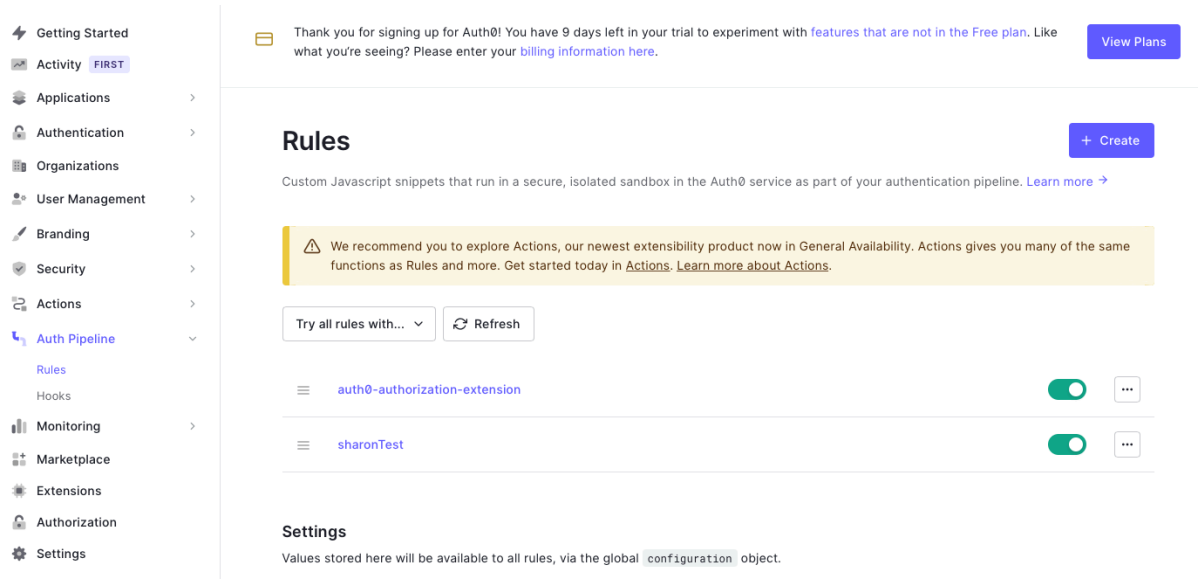


2.2 基于SAML2.0的Auth0与AWS console权限集成

AWS提供了Cognito用于支持数百万用户规模的管理、与第三方社交身份提供商的联合认证，并提供多种方式帮助用户从应用来控制对AWS资源的访问，有关Cognitor的文档和博客已经非常丰富。本博客从通用方案的角度出发，选择了第三方产品Auth0来演示如何通过SAML2.0与AWS控制台实现统一身份权限管理。本文对Ath0与AWS IAM间如何交换SAML Metadata构建信任关系不再赘述，重点讨论如何建立Auth0的用户与IAM角色的映射关系：

- 首先，确定Auth0的用户如何与IAM角色实现映射：
  在RBAC中我们使用用户所属项目以及在项目中担任的角色这两个组合属性来与IAM角色进行映射。
- 然后：扩展Auth0的用户属性：
  缺省状态下，Auth0用户不具备项目、角色等属性，但提供了user_metadata属性实现对用户定义的灵活扩展，演示中我们对该用户添加project、role这两个属性如下图：

- 其次：在Auth0中创建Rule，构建Auth0用户属性与IAM角色的映射关系



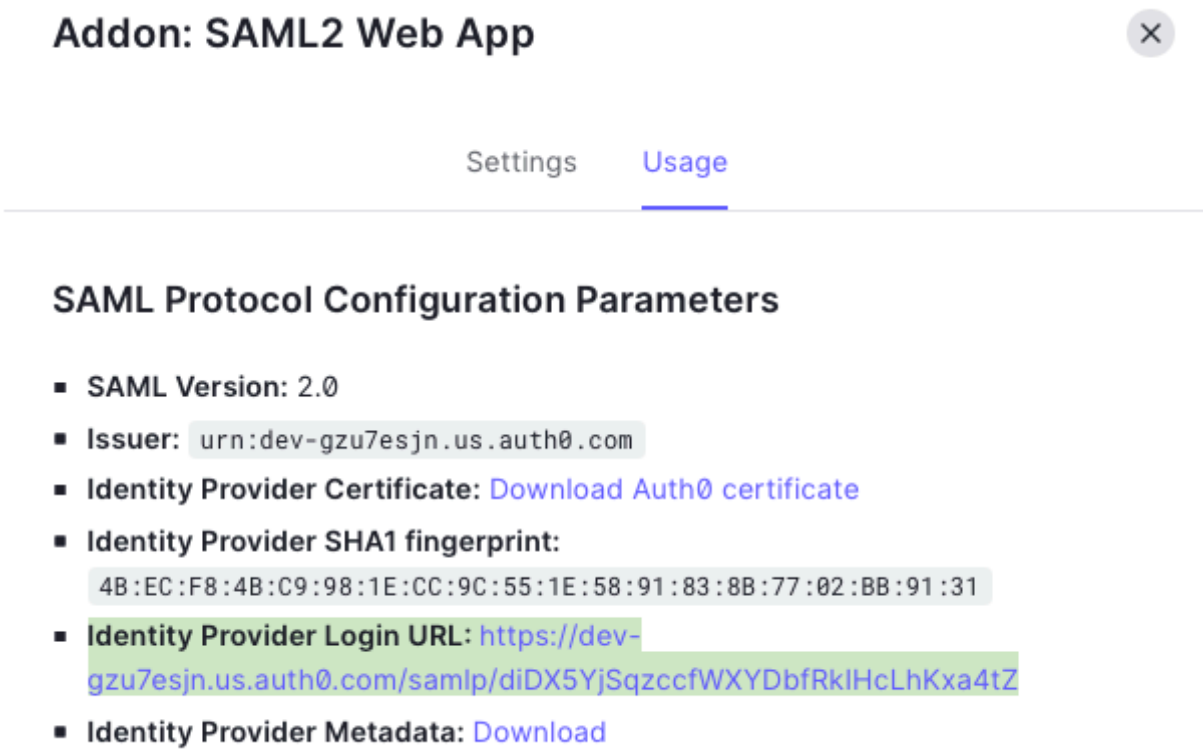
创建客户化的规则（上图中sharonTest）用于实现Auth0的属性与IAM role的映射，如下Javascript代码示例：

```
function (user, context, callback) {
var awsRoles = {
'project1-readonly':
'arn:aws:iam::955513527673:role/auth0saml,arn:aws:iam::955513527673:saml-provider/auth0saml',// AWS中创建的基于SAML2.0联合认证的角色，用于控制project1-manager对资源的访问
```

```
'project2-manager': 'arn:aws:iam::955513527673:role/auth0saml-
readonly,arn:aws:iam::955513527673:saml-provider/auth0saml' // AWS中创建的基
于SAML2.0联合认证的角色，用于控制project2-readonly对资源的访问
};

var userattribute = user.user_metadata.project+'-
'+user.user_metadata.access_role;
user.awsRole = awsRoles[userattribute]; // 根据用户当前所在的groups属性从
awsRoles中匹配对应的角色
user.awsRoleSession = user.name;

context.samlConfiguration.mappings = {
 'https://aws.amazon.com/SAML/Attributes/Role': 'awsRole',
'https://aws.amazon.com/SAML/Attributes/RoleSessionName': 'awsRoleSession'
};
  callback(null, user, context);
}
```

- 最后，登录Auth0提供的IDP URL，身份认证通过后即可跳转到AWS cosole界面

  如下，为在Auth0中构建的基于SAML2.0的application提供的IDP Login URL



输入用户名密码，在Auth0完成身份验证后，自动跳转至AWS console。

从以上实践可以看到，采用RBAC的权限管理模型，关键是如何实现IDP中的用户属性与IAM角色的映射。集成起来简单快速，适合匹配角色数量较少的场景使用。

3. 思路二：基于属性的权限管理模型ABAC

3.1 方案概述

在某些场景，每个项目所使用的资源类型都是相同的，只是不同项目对应不同的资源实例。比如所有项目都只能使用EC2和RDS服务，只是每个项目使用不同的服务实例。对这类同质的项目，就非常适合使用基于ABAC的权限管理模型，通过设定资源的Tag和用户主体的Tag，使用IAM策略中的条件限制进行权限管控。这样可以大幅减少角色的定义工作。

3.2 基于SAML2.0的Auth0与AWS console权限集成

与Auth0结合实现ABAC权限管理模型的方式如下所示：

- 首先，在Auth0中创建Rule，在SAML断言中携带session Tag，将用户的属性作为会话标签进行传递：
  下面的实例代码展示Auth0通过SAML断言将project和role两个属性作为会话标签传递给IAM的PrincipalTag

```
function(user, context, callback) {
  ......
  context.samlConfiguration.mappings = {
    'https://aws.amazon.com/SAML/Attributes/Role': 'awsRole',
    'https://aws.amazon.com/SAML/Attributes/RoleSessionName':
'awsRoleSession',
    'https://aws.amazon.com/SAML/Attributes/PrincipalTag:project':
user.user_metadata.project,
    'https://aws.amazon.com/SAML/Attributes/PrincipalTag:access_role':
user.user_metadata.project
  };
  callback(null, user, context);
}
```

- 然后，在IAM role中构建相应的权限

  下面的示例代码以管控secretsmanager资源为例，展示了权限控制的思路。构建一个role，该role包含的policy覆盖项目中所有使用的资源，并且为每个资源都定义不同角色（如下包含了manager和readonly两个角色）对应的权限。

  SAML断言中携带的的两个session Tag将分别对应IAM角色主体中的aws:PrincipalTag/Project和aws:PrincipalTag/access-role两个主体标签。同时我们需要对AWS 中的资源都按照项目属性打上aws:ResourceTag/project标签。

  这样在policy的condition部分就会根据主体标签（access-role）以及主题标签（project）和资源标签的匹配关系获得所需的权限策略。

```
"Statement": [
{
    "Sid": "AllActionsSecretsManager",
    "Effect": "Allow",
    "Action": "secretsmanager:*",
    "Resource": "*",
    "Condition": {
```

```
      "StringEquals":{
         "aws:ResourceTag/project": "${aws:PrincipalTag/project}",
         "aws:PrincipalTag/access-role": "manager"
      }
    }
},
{
    "Sid": "ReadActionSecretsManager",
    "Effect": "Allow",
    "Action": [ "secretsmanager:Describe*",
  "secretsmanager:Get*",
   "secretsmanager:List*" ],
    "Resource": "*",
    "Condition": {
      "StringEquals":{
         "aws:ResourceTag/project": "${aws:PrincipalTag/project}",
         "aws:PrincipalTag/access-role": "readonly"
          }
     }
}
]
```

基于ABAC的权限模型极大程度简化了policy的维护数量，尤其适用于多租户模式下对资源类型使用相对固定的场景。我们只需要一套policy，使用资源标签和主体标签构建匹配规则就可以覆盖所有权限管控需求，并可以扩展至任意数量的项目。

但ABAC也有自己的限制。比如 IAM 角色的附加托管策略数量具有上限（20）， 每个托管策略的字符大小也具有上限（6144字节），对构建较为复杂的策略时可能会受到相应的限制；并且ABAC依赖资源标签进行细粒度的资源隔离，这就需要对资源都打上标签且要求标签准确 ，对有些不提供ResourceTag的资源（如S3 prefix）ABAC的方式就会受限。

BLOG2

构建基于Auth0的统一身份管控平台实现AWS控制台的登录与授权

-- 第二篇 基于动态策略生成的权限管控模型

越来越多的企业在迁移上云旅程中都会面临多云的选择。不管是出于主动原因希望融合多云的服务优势，还是出于被动原因需要适应公司的业务并购，抑或是因为上云过程中需要实现混合云架构的打通，构建多云的统一身份权限管控平台往往是这些企业需要解决的问题。

构建这一平台，不仅要实现单点登录，还要考虑统一身份管控、统一权限管控和云上用户行为审计；并且企业内人员的多组织管理模型通常与云上基于角色的IAM管理模型存在较大差异，这进一步增加了构建跨云身份管理平台的难度。

本博客从企业客户的实际需求和技术难点出发，分别阐述了使用

- RBAC -- 基于角色的权限管控
- ABAC -- 基于属性的权限管控
- 动态策略生成

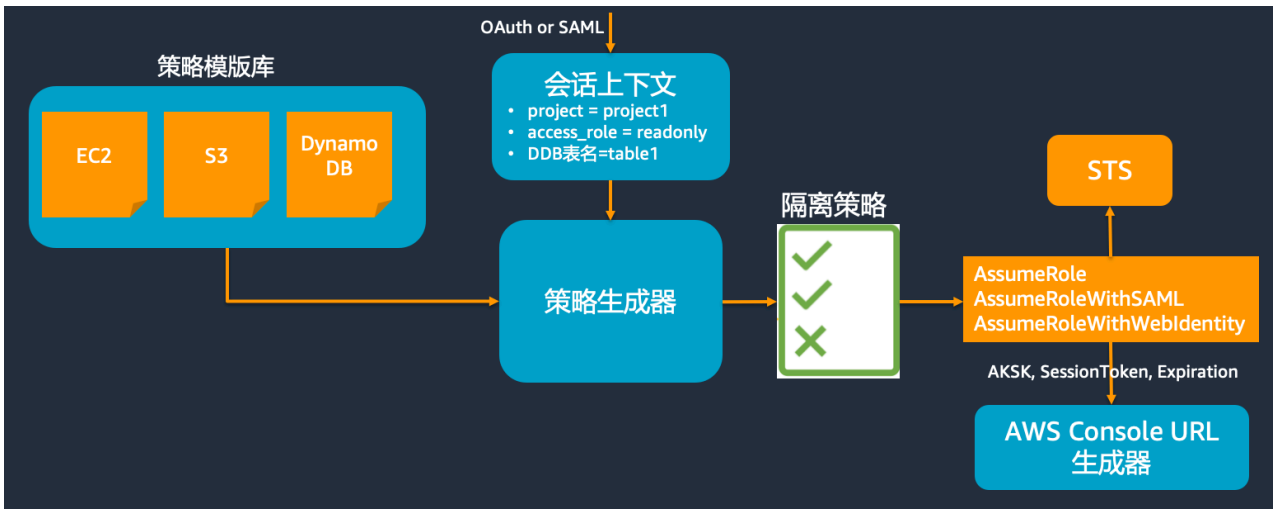
这三种思路实现企业统一身份管控平台，并与公有云控制台(Web console)实现身份和权限的打通；以及三种思路各自的适用场景；并在此基础上给出了基于Auth0构建该平台，完成与AWS控制台集成，实现统一登录与授权的具体实现代码样例。

受篇幅所限，我们把博客分成两部分，上一篇讲述RBAC和ABAC的实现方法；本篇主要讲述动态策略生成的实现方法。

1. 方案概述

在第一篇，我们讲述的RBAC思路用于实现IDP中的用户属性（项目与角色的组合）与IAM角色的一对一映射。集成起来简单快速，适合用户的项目x角色组合数量较少的使用场景；而ABAC则适用于多租户模式下对资源类型使用相对固定的场景。我们使用一个policy，通过资源标签和主体标签的匹配规则就可以覆盖所有权限管控需求，并可以扩展至任意数量的项目

但实际的企业环境中，如果项目的数量很多，而且每个项目中用到的资源类型又不尽相同，RBAC和APBAC都很难满足需求。为了提升环境的可管理性和敏捷性，避免可能产生的策略数量和尺寸的爆炸性增长，基于动态策略生成的权限管理模型将是一个更为通用的解决方案。基本思路如下图：



- 首先，构建策略模版库，用于保存各种云资源的动态权限模版
  模版库可以保存在DynamoDB或S3中。为便于后续的维护和管理，建议对每个资源的不同角色分别维护一个独立的模版。下面的策略模版库演示了允许查询和启动EC2实例的一个策略模版，其中{{region}}、{{accountid}}和{{project}}都为模版中的placeholder，以备后续动态生成策略。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:Describe*",
                "ec2:List*"
```

```
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:StartInstances"
            ],
            "Resource": "arn:aws:ec2:{{region}}:{{accountid}}:instance/*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/environment": "{{project}}"
                }
            }
        }
    ]
}
```

- 之后，维护用户属性与策略模版间的对应关系
  在实际项目中，该对应关系可以通过开发配置界面完成，便于管理员进行动态配置。

项目	角色	模版列表
Project1	Readonly	EC2-ReadOnly-template
Project1	Readonly	S3-ReadOnly-Bucket-template
Project1	Readonly	DDB-ReadOnly-Table-template
Project1	Manager	EC2-AllAccess-template

- 最后，构建策略生成器，用于根据用户登录后的上下文信息动态生成IAM策略
  用户在IDP完成身份认证后，通过OpenID或SAML断言携带用户的上下文信息。我们构建的策略动态生成器需要先验证用户的合法身份，然后根据上下文信息获取所需的策略模版列表，并生成所需的策略，之后通过AWS STS服务完成AssumeRole，获得临时的AKSK、SessionToken等信息，生成访问AWS console的URL，以保证资源访问的隔离。

这一方案的好处显而易见，它不拘于使用RBAC或ABAC模型，也不依赖于标签的使用，我们可以在模版中任何需要的地方放置placeholder构建所需策略，非常灵活。

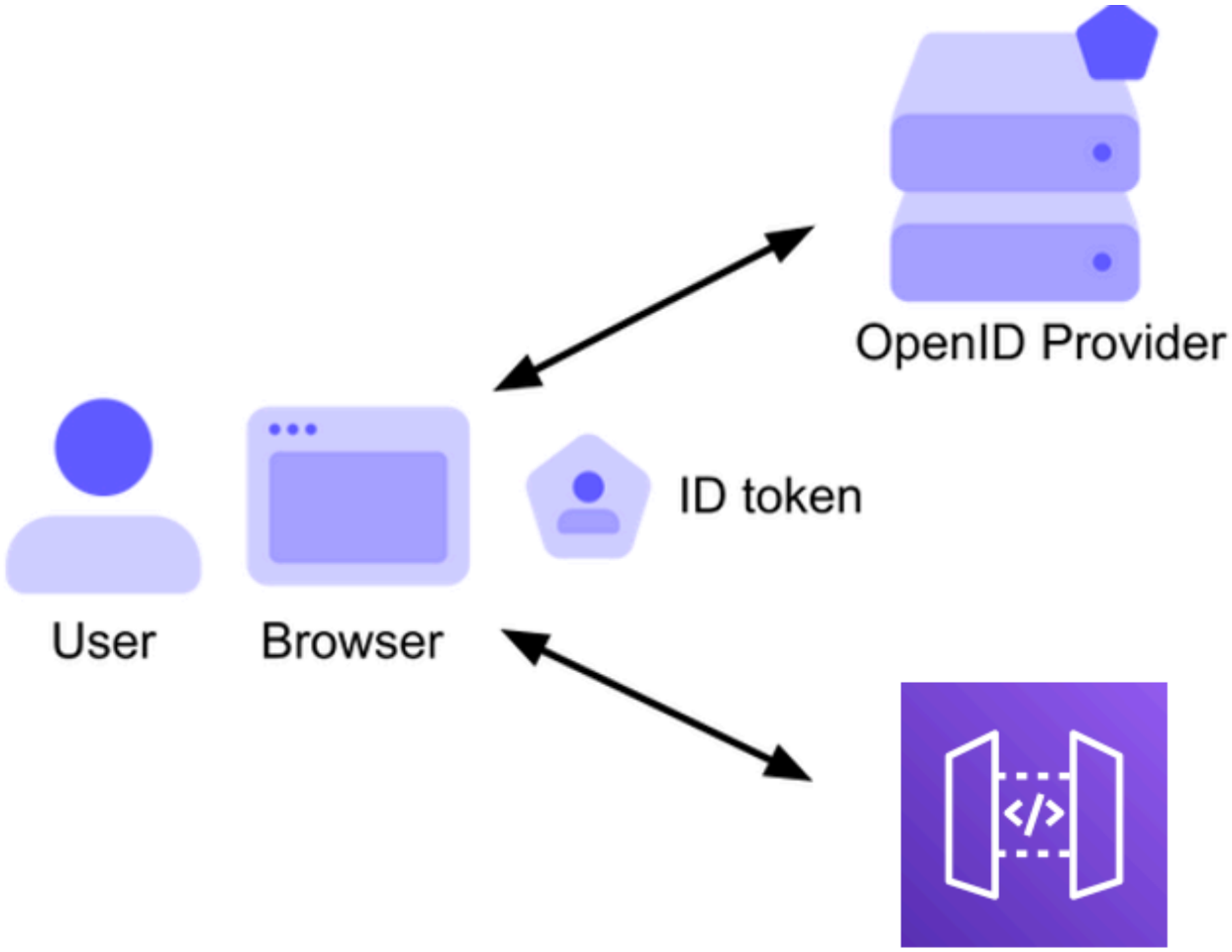
下面我们对几个技术关键点的实现做详细展开：

2. 配置Auth0实现基于OIDC的身份认证

本博客，我们使用Auth0作为统一身份管控平台的IDP。验证中，我们使用了OIDC的身份层协议，在Auth0完成身份认证后将向管控平台返回IDToken和Access Token。

2.1 IDToken or AccessToken?

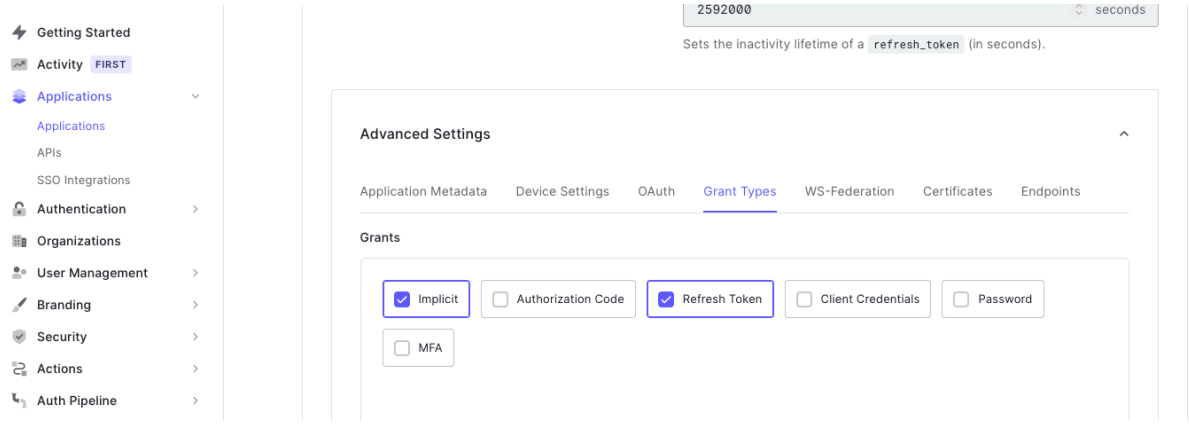
当前的场景是企业的身份管控平台需要从Auth0获得IDToken，表明已完成身份认证，具体的资源授权由API GW和后端的Lambda根据Token中携带的用户属性自行完成，因此应向后端的APIGateway传递IDtoken。在其他的场景，如果是访问Auth0获得对资源的访问授权，携带允许访问的资源scope信息到resource server获取相应资源的场景（如常见的使用微信认证，并允许获得微信头像的场景），则应使用access token。



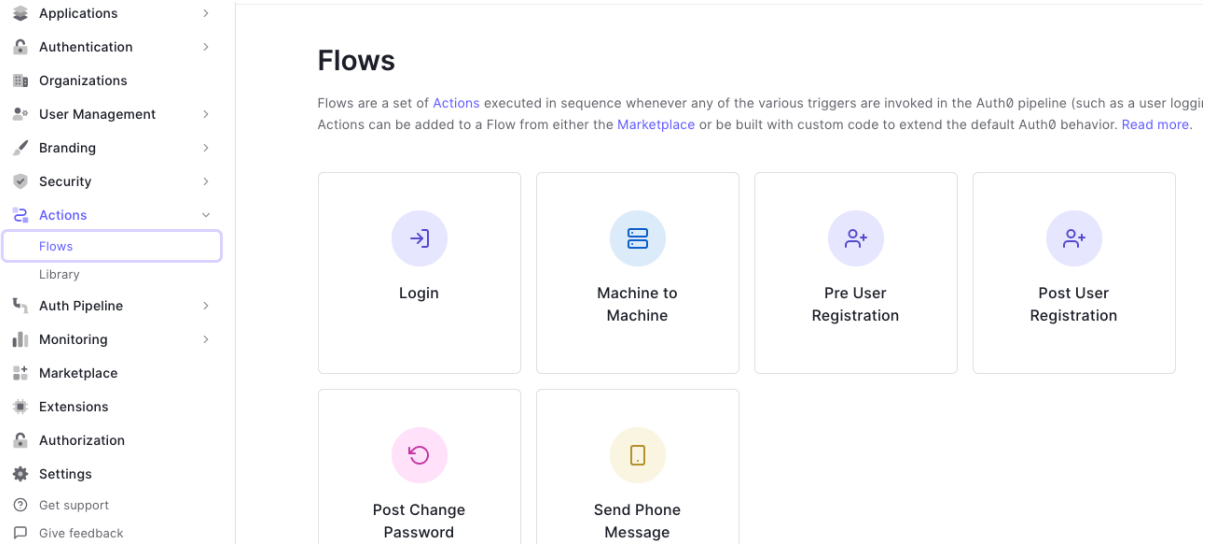
2.2 配置Auth0获取IDToken

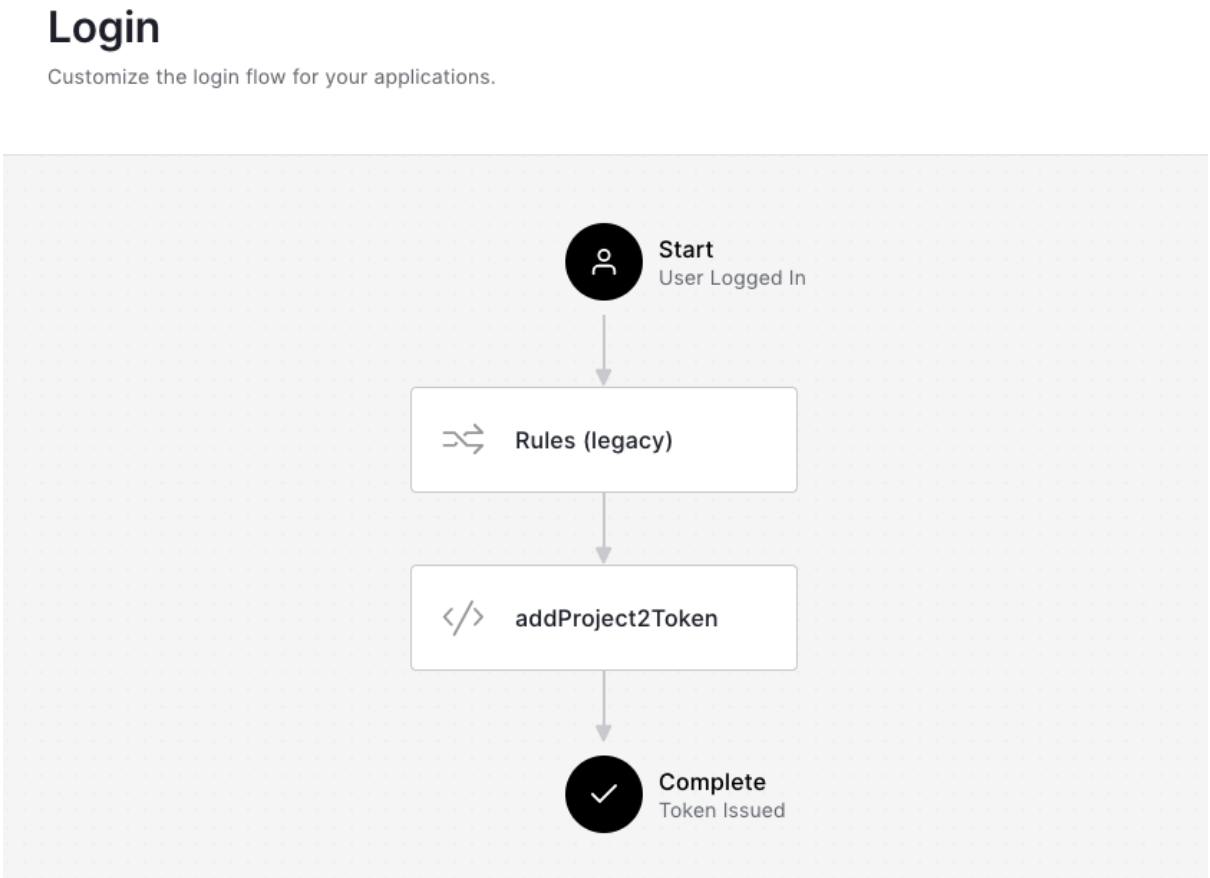
在这个场景中，企业的身份管控平台需要依赖Auth0完成身份认证，再将从Auth0获得的包含用户属性信息的ID token携带在http header中，请求API Gateway调用。在Auth0中我们需要完成如下的配置，以获得包含完整用户属性信息的ID token：

- 首先，在Auth0中创建API application
  创建成功后，将获得application对应的clientID和client secret。clientID后面会包含在ID token的aud中，用于表明该token是合法颁发给该application的。
- 然后，在创建的application中选择认证使用的工作流
  在application ->新创建的Application ->Advanced Settings中选择Implicit的授权模式。该模式为OAuth2.0中的RFC6749中的隐式授权模式，授权过程较为简单，不需授权码可直接获得IDToken。



- 最后，修改application的登录流程，加入“appProject2Token”的action





该Action用于把用户的user_metadata中的project和role的属性信息加入到IDToken中：

```
exports.onExecutePostLogin = async (event, api) => {
  const namespace = 'https://mytesttest.com';
  const project = event.user.user_metadata.project;
  const role = event.user.user_metadata.role;

  //if (event.authorization) { 这里为简便起见忽略了验证部分，实际项目中可根据安全需要对applications ->api -> permission中限定的scope进行验证，允许访问相应的内容时，才将用户的相关信息放入IDToken
    // Set claims
    api.idToken.setCustomClaim(`${namespace}/project`, event.user.user_metadata.project);
    api.idToken.setCustomClaim(`${namespace}/role`, event.user.user_metadata.role);
  //}
};
```

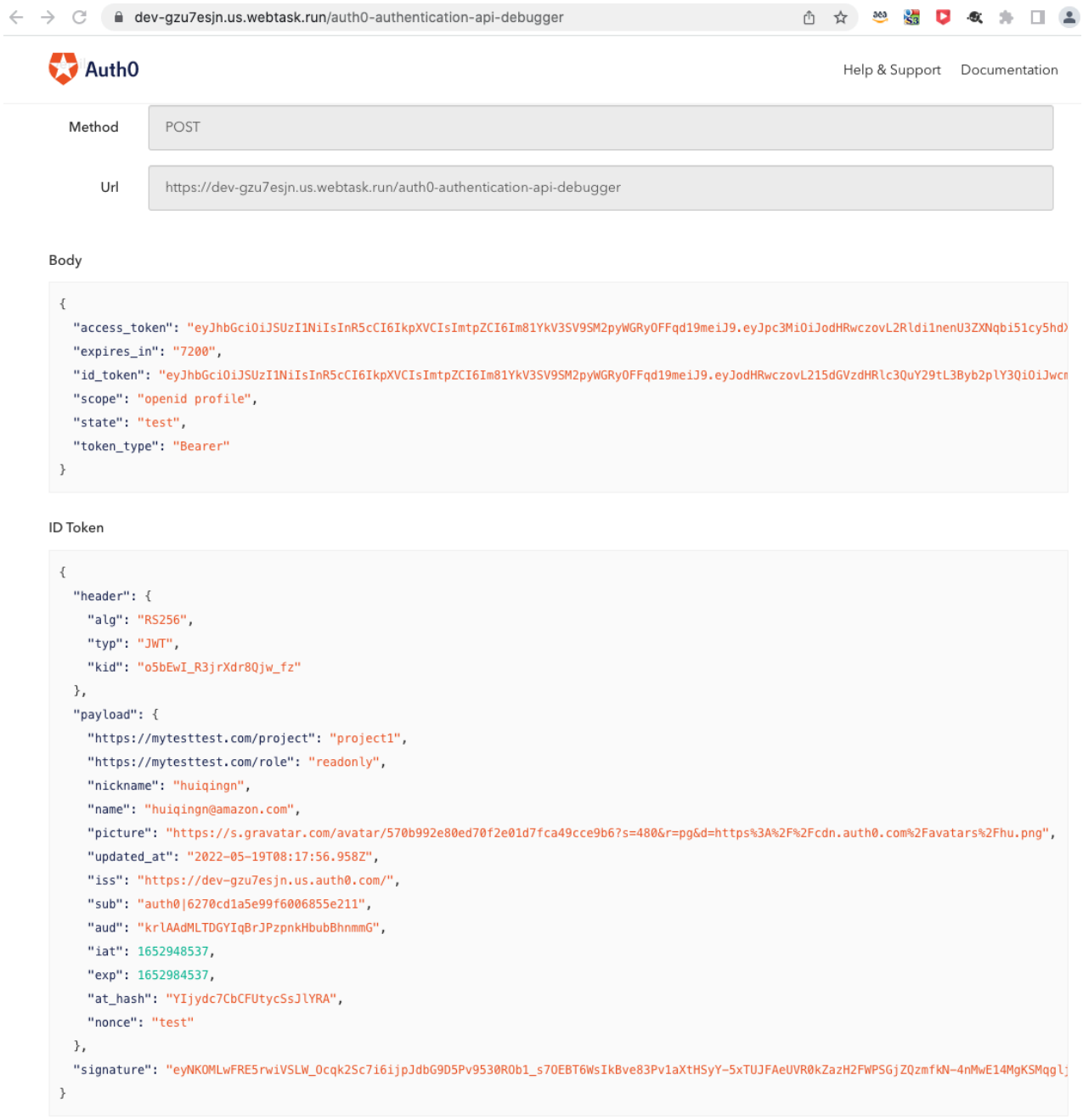
2.3 IDToken获取验证

下面我们验证从Auth0中获取的IDToken

Auth0提供了Authentication API Debugger可以方便地查看获取到的IDToken。对创建的API application使用如下的authorizer接口进行访问（下面的{{}}部分用API application产生的domain和clientid替换），rediect_url为Authentication API Debugger的URL。

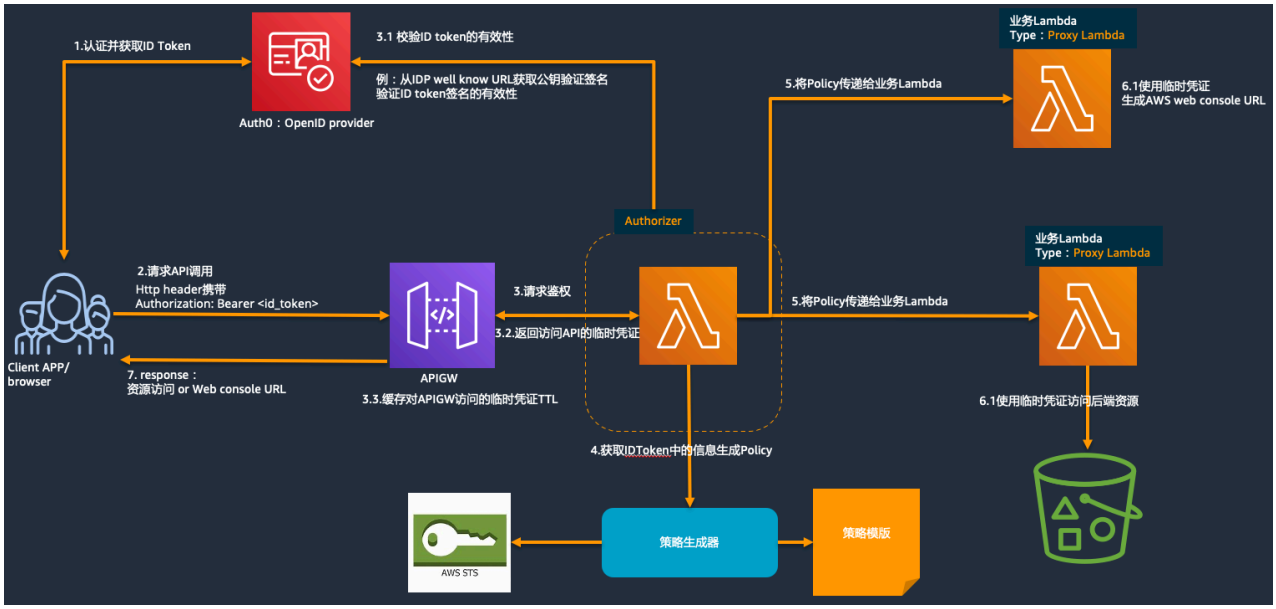
https://{{ApplicationDomain}}/authorize?response_type=id_token token&response_mode=form_post&client_id={{CLINTID}}&redirect_uri=https://dev-gzu7esjn.us.webtask.run/auth0-authentication-api-debugger&scope=openid profile&state=test&nonce=test&audience=https://oauth0api.com

可以看到返回的OpenID中包含我们之前定义的用户属性信息（project和role）。



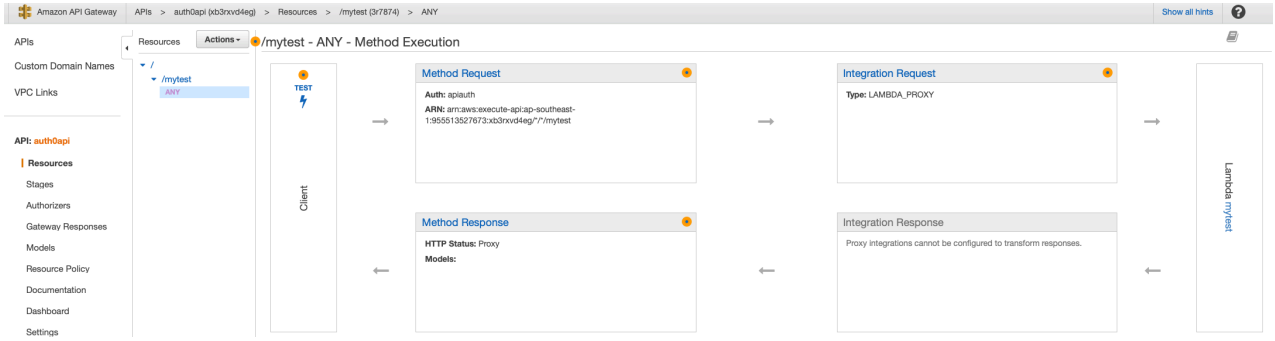
3.动态策略生成器实现

在实际项目中，我们使用AWS API Gateway的Lambda Authorizer构建了策略生成器。Lambda Authorizer是API Gateway的一个功能，该功能使用 Lambda 函数来控制对 API Gateway 的访问。在该Lambda函数中我们主要完成了两项工作：首先，根据持有者令牌（如 OAuth 或 SAML）验证用户身份的合法性，验证通过即可允许API Gateway的调用；同时对合法用户，根据用户的上下文信息结合策略模版动态生成访问策略，并将生成的策略传递给后端的业务Lambda，业务Lambda根据这些policy通过AssumeRole获得临时凭证来完成资源权限的隔离。

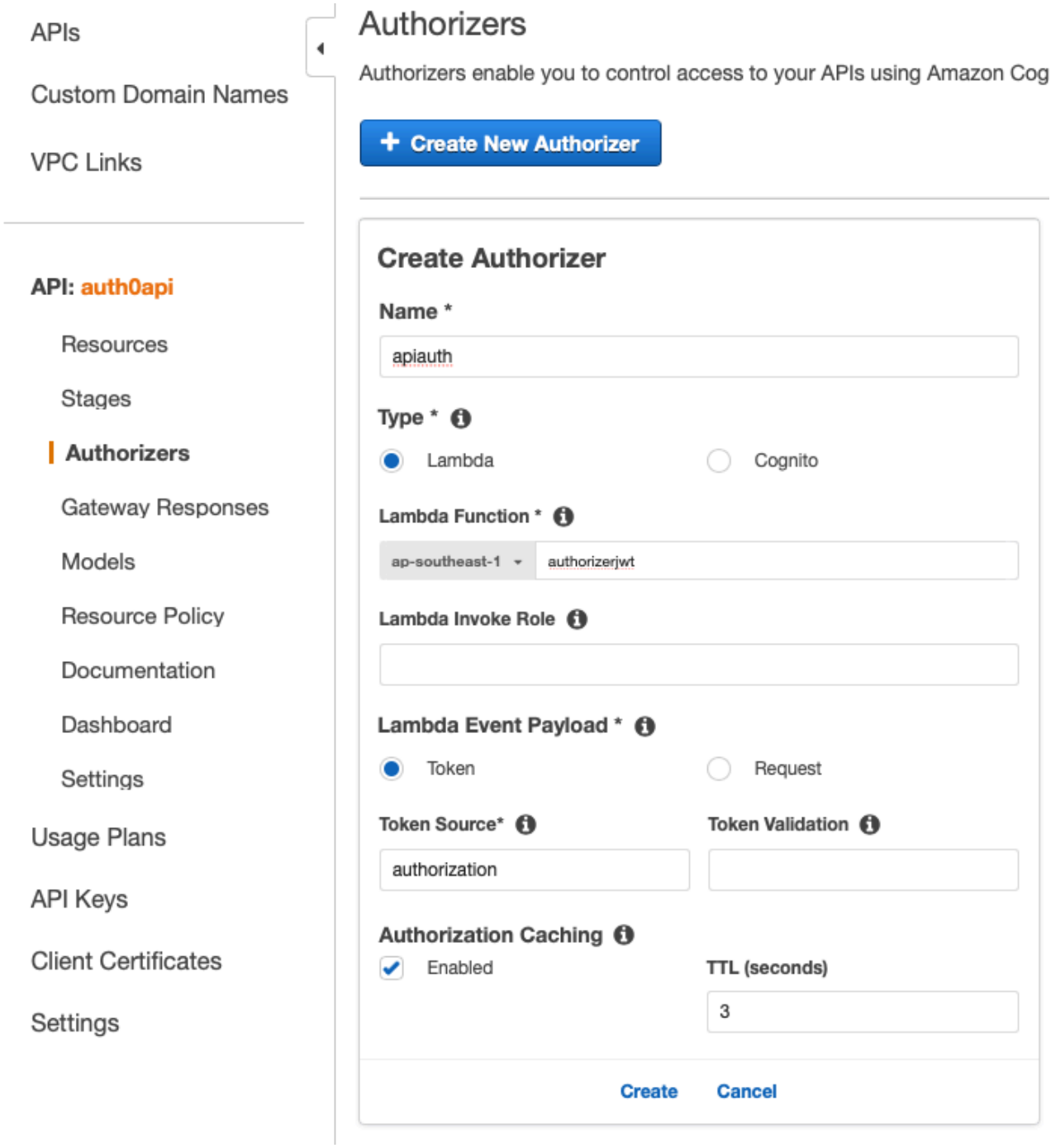


3.1构建具有Authorizer Lambda的APIGateway

实现中，我们构建了名为“auth0api”的Rest APIGateway，调用后端名为“mytest”的业务Lambda，业务Lambda生成访问aws console的URL通过APIGateway返回给前端的client app/browser



为该APIGateway添加名为“apiauth”的Authorizer，对应的Lambda function为自建的“authorzerjwt”，选择Token作为Lambda Event负载，选择认证后的结果信息在APIGateway中保留3s，期间不需要再次通过“authorzerjwt”做认证



3.2 Authorizer Lambda的实现

该函数是动态策略生成器的核心部分。它负责获取APIGateway中的IDToken信息，校验IDToken的合法性。

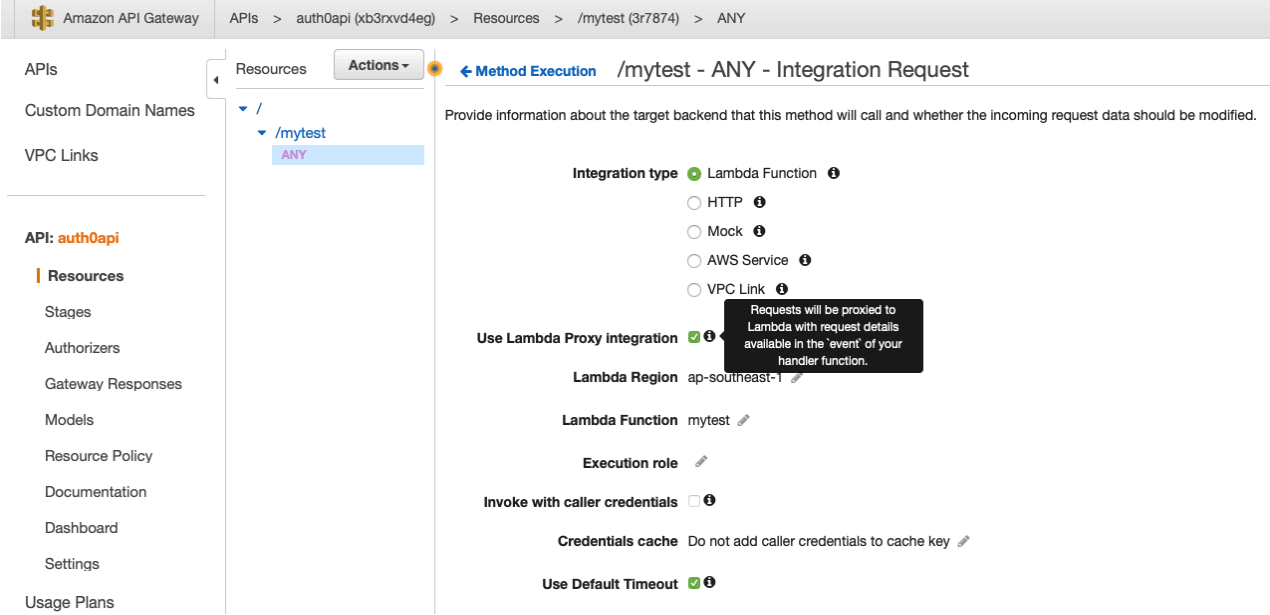
```
def lambda_handler(event, context):
    #从Event中获取APIGateway中传来的IDToken
    token = event['authorizationToken'].split(" ")
    if (token[0] != 'Bearer'):
        raise Exception('Authorization header should have a format Bearer <JWT>
Token')
    jwt_bearer_token = token[1]
    unauthorized_claims = jwt.get_unverified_claims(jwt_bearer_token)
    #并从IDToken中获得project和role的属性
```

```
    project_attri = unauthorized_claims['https://mytesttest.com/project']
    accessrole_attri = unauthorized_claims['https://mytesttest.com/role']

    #从Auth0中创建的API Application中获得jwt签名的公共密钥
    keys_url = 'https://dev-gzu7esjn.us.auth0.com/.well-known/jwks.json'
    with urllib.request.urlopen(keys_url) as f:
        response = f.read()
    keys = json.loads(response.decode('utf-8'))['keys']

    #用公共密钥校验IDToen中的签名部分，并核对IDToken中的aud是否为Auth0中创建的API Application对应的ClientID，是则认为用户校验通过。该部分代码篇幅原因删略，如需要请参见github链接
    response = validateJWT(jwt_bearer_token, appclientid, keys)

    #get authenticated claims
    if (response == False):
        raise Exception('Unauthorized')
    else:
        principal_id = response["sub"]

    #生成APIGateway的policy--authResponse，通过Authorizer lambda 返回给apigw，决定是否允许调用APIGateway
    tmp = event['methodArn'].split(':')
    api_gateway_arn_tmp = tmp[5].split('/')
    aws_account_id = tmp[4]
    policy = AuthPolicy(principal_id, aws_account_id)
    policy.restApiId = api_gateway_arn_tmp[0]
    policy.region = tmp[3]
    policy.stage = api_gateway_arn_tmp[1]
    policy.allowAllMethods()
    authResponse = policy.build()

    #同时，获取IDToken中的用户属性信息，结合策略模版为后端的业务lambda生成动态policy：deliverpolicy，通过上下文传给后端业务lambda
    deliverpolicy = policyGenerator(project_attri, accessrole_attri)
    ##将生成的动态policy向后端业务传递
    context = {
        'policy': deliverpolicy
    }
    authResponse['context'] = context
    return authResponse

## 根据用户的属性信息从DynamoDB中取出策略模版，使用策略模版动态生成policy
def policyGenerator(project, projrole):
    dynamodb = boto3.client('dynamodb')
    response = dynamodb.get_item(
        TableName="policytemplate",
        Key={
        'project': {'S': 'project'},
```

```
        'role': {'S': 'projrole'}
        }
    )
    thecontext = {'region':REGIONID,'accountid':ACCOUNTID,'project':project}
    temppolicy = pystache.render(response['Item']['policy']['S'],thecontext)
    return temppolicy
```

3.3 构建业务Lambda生成符合权限管控的AWS Console

Athorizer Lambda生成了APIGateway的访问策略和后端业务所需的动态策略，返回给APIGateway。后端业务Lambda与APIGateway的集成注意使用“Lambda proxy Integration”，以便从Lambda的event参数中获取APIGateway传递的从Athorizer Lambda获得的动态policy




```
def lambda_handler(event, context):
  ##由于AWS cosole不支持角色串联（参见https://aws.amazon.com/cn/premiumsupport/knowledge-center/iam-role-chaining-limit/），因此示例中使用了AKSK再通过assumerole的方式使用policy
    session = boto3.session.Session(aws_access_key_id="xxxxx", aws_secret_access_key="xxxx")
    sts_connection = session.client('sts')
    ##在APIGateway传递来的event中获取由authorizer lambda生成的动态policy
    thepolicy = event['requestContext']['authorizer']['policy']

    ##由于assume_role最终生成的临时凭证是基础role_arn和policy的合集，因此示例中选用的较大权限。实际生产中可以根据需要选择合理范围。同时注意对role_arn创建对指定aksk的用户的信任策略，以便assumerole具有权限。
    role_arn = "arn:aws:iam::{}:role/adminrole".format(aws_account_id)
    assumed_role = sts_connection.assume_role(
        RoleArn=role_arn,
        RoleSessionName="console-session",
        Policy=thepolicy
    )
    credentials = assumed_role["Credentials"]
```

```
    # 获得临时凭证的AKSK和session Token
    url_credentials = {}
    url_credentials['sessionId'] = credentials['AccessKeyId']
    url_credentials['sessionKey'] = credentials['SecretAccessKey']
    url_credentials['sessionToken'] = credentials["SessionToken"]
    json_string_with_temp_credentials = json.dumps(url_credentials)

    #用临时凭证构建aws console
    request_parameters = "?Action=getSigninToken"
    request_parameters += "&SessionDuration=43200"
    if sys.version_info[0] < 3:
        def quote_plus_function(s):
            return urllib.quote_plus(s)
    else:
        def quote_plus_function(s):
            return urllib.parse.quote_plus(s)

    request_parameters += "&Session=" +
quote_plus_function(json_string_with_temp_credentials)
    request_url = "https://signin.aws.amazon.com/federation" +
request_parameters
    r = requests.get(request_url)
    signin_token = json.loads(r.text)

    request_parameters = "?Action=login"
    request_parameters += "&Issuer=Example.org"
    request_parameters += "&Destination=" +
quote_plus_function("https://console.aws.amazon.com/")
    request_parameters += "&SigninToken=" + signin_token["SigninToken"]
    request_url = "https://signin.aws.amazon.com/federation" +
request_parameters

    return {
        'statusCode': 200,
        'body': json.dumps(request_url)
    }
```

4. 验证

使用2.3节获得的IDToken调用APIGateway

curl --request GET --url https://{{APIID}}.execute-api.ap-southeast-1.amazonaws.com/default/mytest --header 'authorization: Bearer {{IDToken}}'

即可获得访问aws console的url如下：

https://signin.aws.amazon.com/federation?Action=login&Issuer=Example.org&Destination=https%3A%2F%2Fconsole.aws.amazon.com%2F&SigninToken=xxxx

该URL的有效期可以在assume_role中设定从15min到12小时的时长。

总结

至此，我们介绍了基于RBAC、ABAC和动态策略生成3中方式实现统一身份认证平台中用户属性与IAM角色的对应，从而完成统一身份管控平台与AWS控制台的SSO和统一授权。

实际场景中，如果用户多种属性的组合数量非常有限，推荐使用RBAC实现属性与IAM角色的直接映射，简单快捷；如果不同属性的用户使用的资源类型整齐划一，则可以使用ABAC通过会话标签和资源标签实现IAM角色的限定；但如果用户多种属性的组合过于复杂，而且不同类型的用户使用的资源需求有不尽相同，则可以使用动态策略生成的方式，通过策略模版生成动态策略，实现灵活的权限隔离。