



Amazon Cognito

# Components of a Serverless app

Logic



Lambda

State



RDS



DynamoDB



Amazon S3



Cognito

Communication



API Gateway



Kinesis



SNS



SQS

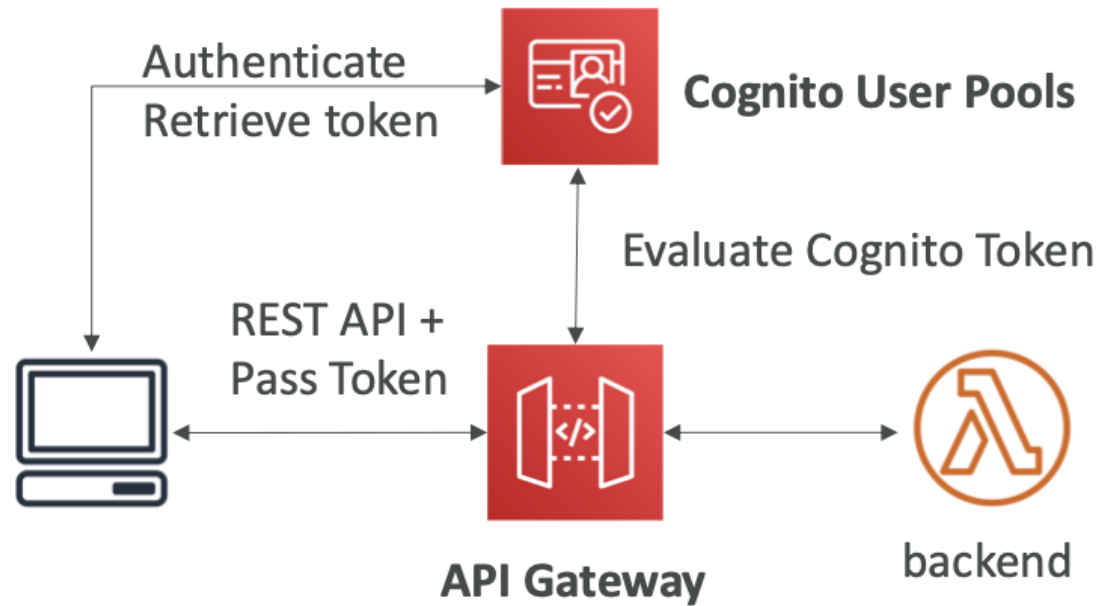
# Amazon Cognito

- We want to give users an identity so that they can interact with our application.
- Cognito User Pools:
  - Sign in functionality for app users.
  - Integrate with API Gateway & Application Load Balancer.
- Cognito Identity Pools (Federated Identity):
  - Provide AWS credentials to users so they can access AWS resources directly.
  - Integrate with Cognito User Pools as an identity provider.
- Cognito vs IAM: “hundreds of users”, “mobile users”.

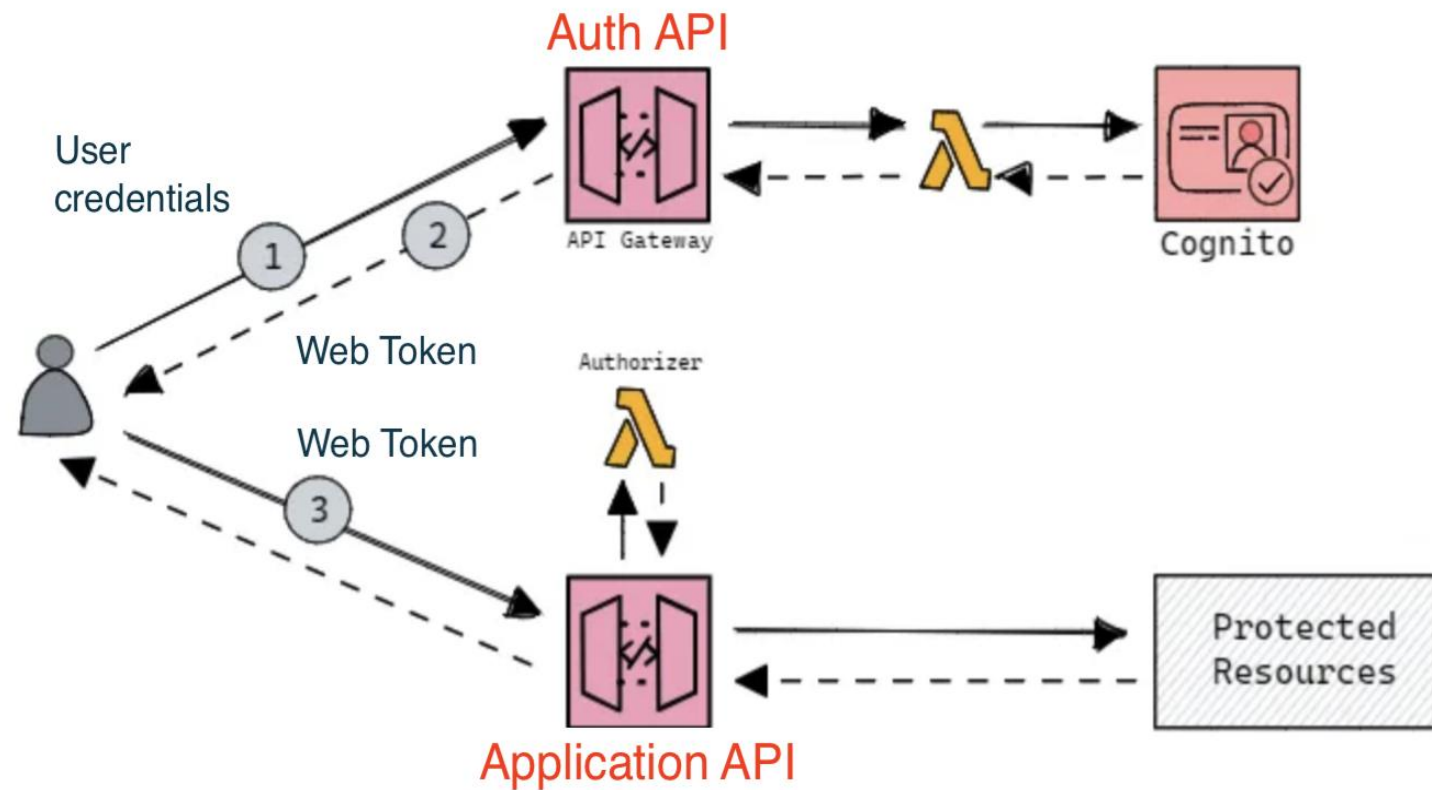
# Cognito User Pool.

- Creates a serverless database of user for your web & mobile apps.
- Simple login: Username (or email) / password combination.
- Password reset.
- Email & Phone Number Verification.
- Multi-factor authentication (MFA).
- Federated Identities: Facebook, Google...
- Feature: block users if their credentials are compromised elsewhere
- Include JSON Web Token (JWT) in Login response.

# Cognito's role



# Typical architecture



# Demo – User pool

```
109
110 // CDK setup
111 const userPool = new UserPool(this, 'UserPool', {
112   signInAliases: { username: true, email: true },
113   selfSignUpEnabled: true,
114   removalPolicy: RemovalPolicy.DESTROY,
115 });
116
117 const appClient = userPool.addClient('AppClient', {
118   authFlows: { userPassword: true },
119 });
120
```

The screenshot shows the AWS Management Console interface for Amazon Cognito. The left sidebar contains the 'Amazon Cognito' header and a link to 'User pools'. The main content area is titled 'User pools (1)' and includes a 'Create user pool' button. Below this is a table with the following data:

User pool name	User pool ID	Created time	Last updated time
UserPool6D0DFADB-9PuJm6L1zKAN	eu-west-1_0tVi4i4ET	9 minutes ago	9 minutes ago

# Demo – APIs.

## Auth API

- ▼ /
  - OPTIONS
- ▼ /auth
  - OPTIONS
- ▼ /auth/confirm\_signup
  - OPTIONS
  - POST
- ▼ /auth/current\_user
  - GET
  - OPTIONS
- ▼ /auth/signin
  - OPTIONS
  - POST
- ▼ /auth/signout
  - GET
  - OPTIONS
- ▼ /auth/signup
  - OPTIONS
  - POST

## Application API

- ▼ /
  - OPTIONS
- ▼ /protected
  - GET
  - OPTIONS
- ▼ /public
  - GET
  - OPTIONS

```
λ lambda
  λ auth
    TS authorizer.ts
    TS confirm-signup.ts
    TS current-user.ts
    TS signin.ts
    TS signout.ts
    TS signup.ts
    TS protected.ts
    TS public.ts
```



# Demo – Sign up

The screenshot displays a REST client interface with a POST request to the endpoint `https://87jqug6skd.execute-api.eu-west-1.amazonaws.com/prod/auth/signu...`. The request body is a JSON object with the following fields: `username` (value: "userA"), `password` (value: "passABCDE!2"), and `email` (value: "o[redacted]@gmail.com"). An orange arrow points to the password field. The response is a 200 OK status with a response time of 1630 ms and a body size of 613 B. The response body is shown in JSON format, containing a `message` object with `$metadata` details: `httpStatusCode` (200), `requestId` ("ad8071ef-a53a-4df9-8b0c-c96fb5631bbe"), and `attempts` (1).

POST ▼ `https://87jqug6skd.execute-api.eu-west-1.amazonaws.com/prod/auth/signu...` Send ▼

Params Auth Headers (9) Body ● Pre-req. Tests Settings Cookies

raw ▼ Text ▼

```
1 {
2   "username": "userA",
3   "password": "passABCDE!2",
4   "email": "o[redacted]@gmail.com"
5 }
```

Body ▼ 🌐 200 OK 1630 ms 613 B Save Response ▼

Pretty Raw Preview Visualize JSON ▼ 📄 🔍


```
1 {
2   "message": {
3     "$metadata": {
4       "httpStatusCode": 200,
5       "requestId": "ad8071ef-a53a-4df9-8b0c-c96fb5631bbe",
6       "attempts": 1,

```

📁 Bootcamp 📄 Runner 🗑️ Trash

# Demo – SignUp

```
129
130 const client = new CognitoIdentityProviderClient({ region: "eu-west-1" });
131
132 export const handler: APIGatewayProxyHandlerV2 = async (event) => {
133
134     const { username, email, password }: eventBody = JSON.parse(event.body);
135
136     const params: SignUpCommandInput = {
137         ClientId: process.env.CLIENT_ID!,
138         Username: username,
139         Password: password,
140         UserAttributes: [{ Name: "email", Value: email }],
141     };
142
143     try {
144         const command = new SignUpCommand(params);
145         const res = await client.send(command);
146         return {
147             statusCode: 200,
148             body: JSON.stringify({
149                 message: res,
150             }),
151         };
152     } catch (err) {....}
153 }
```



# Demo – Confirm SignUp

The screenshot displays a REST client interface with a POST request to `https://87jqug6skd.execute-api.eu-west-1.amazonaws.com/prod/auth/confirm_signup`. The request body is a JSON object: `{ "username": "userA", "code": "105002" }`. The response is a 200 OK status with a response time of 864 ms and a body size of 374 B. The response body is shown in a pretty-printed JSON format: `{ "message": "User userA successfully confirmed", "confirmed": true }`. Two orange arrows highlight the mapping from the request body fields to the response fields.

**Request:**

- Method: POST
- URL: `https://87jqug6skd.execute-api.eu-west-1.amazonaws.com/prod/auth/confirm_signup`
- Body (Text):

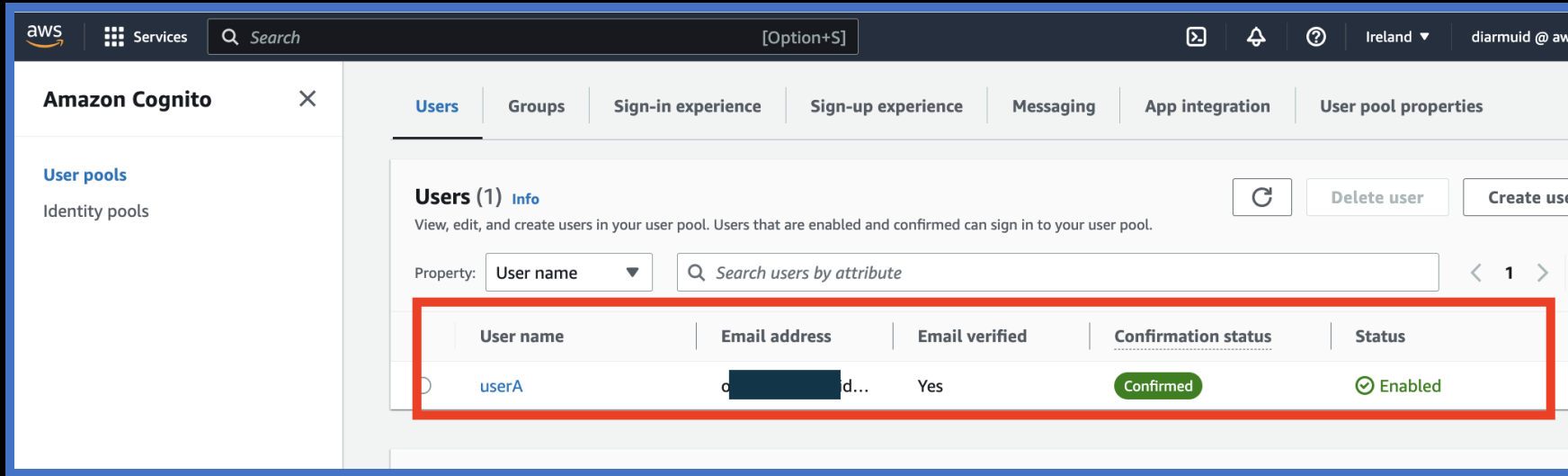
```
1 {
2   "username": "userA",
3   "code": "105002"
4 }
```

**Response:**

- Status: 200 OK
- Time: 864 ms
- Size: 374 B
- Body (Pretty):

```
1 {
2   "message": "User userA successfully confirmed",
3   "confirmed": true
4 }
```

# Demo – Confirm SignUp



The screenshot shows the Amazon Cognito console interface. The top navigation bar includes the AWS logo, a 'Services' menu, a search bar, and a user profile 'diarmuid @ aw'. The left sidebar shows 'Amazon Cognito' with a close button and a list of 'User pools' and 'Identity pools'. The main content area has tabs for 'Users', 'Groups', 'Sign-in experience', 'Sign-up experience', 'Messaging', 'App integration', and 'User pool properties'. The 'Users' tab is active, displaying 'Users (1)' with an 'Info' link. Below this is a description: 'View, edit, and create users in your user pool. Users that are enabled and confirmed can sign in to your user pool.' There is a 'Property' dropdown set to 'User name' and a search bar 'Search users by attribute'. A table of users is shown, with one user 'userA' highlighted by a red box. The table has columns for 'User name', 'Email address', 'Email verified', 'Confirmation status', and 'Status'. The user 'userA' has an email address starting with 'd...', is verified, has a 'Confirmed' status, and is 'Enabled'.

User name	Email address	Email verified	Confirmation status	Status
userA	d...	Yes	Confirmed	Enabled

# Demo – Confirm SignUp

```
157 const client = new CognitoIdentityProviderClient({ region: "eu-west-1" });
158
159 type eventBody = { username: string; code: string };
160
161 export const handler: APIGatewayProxyHandlerV2 = async (event) => {
162
163     const { username, code }: eventBody = JSON.parse(event.body);
164
165     const params: ConfirmSignUpCommandInput = {
166         ClientId: process.env.CLIENT_ID!,
167         Username: username,
168         ConfirmationCode: code,
169     };
170
171     try {
172         const command = new ConfirmSignUpCommand(params);
173         const res = await client.send(command);
174
175         return {
176             statusCode: 200,
177             body: JSON.stringify({
178                 message: `User ${username} successfully confirmed`,
179                 confirmed: true,
180             }),
181         };
182     } catch (err) { ... }
183 };
184
```

You, 1 second ago • Uncommitted changes

# Demo – Sign In

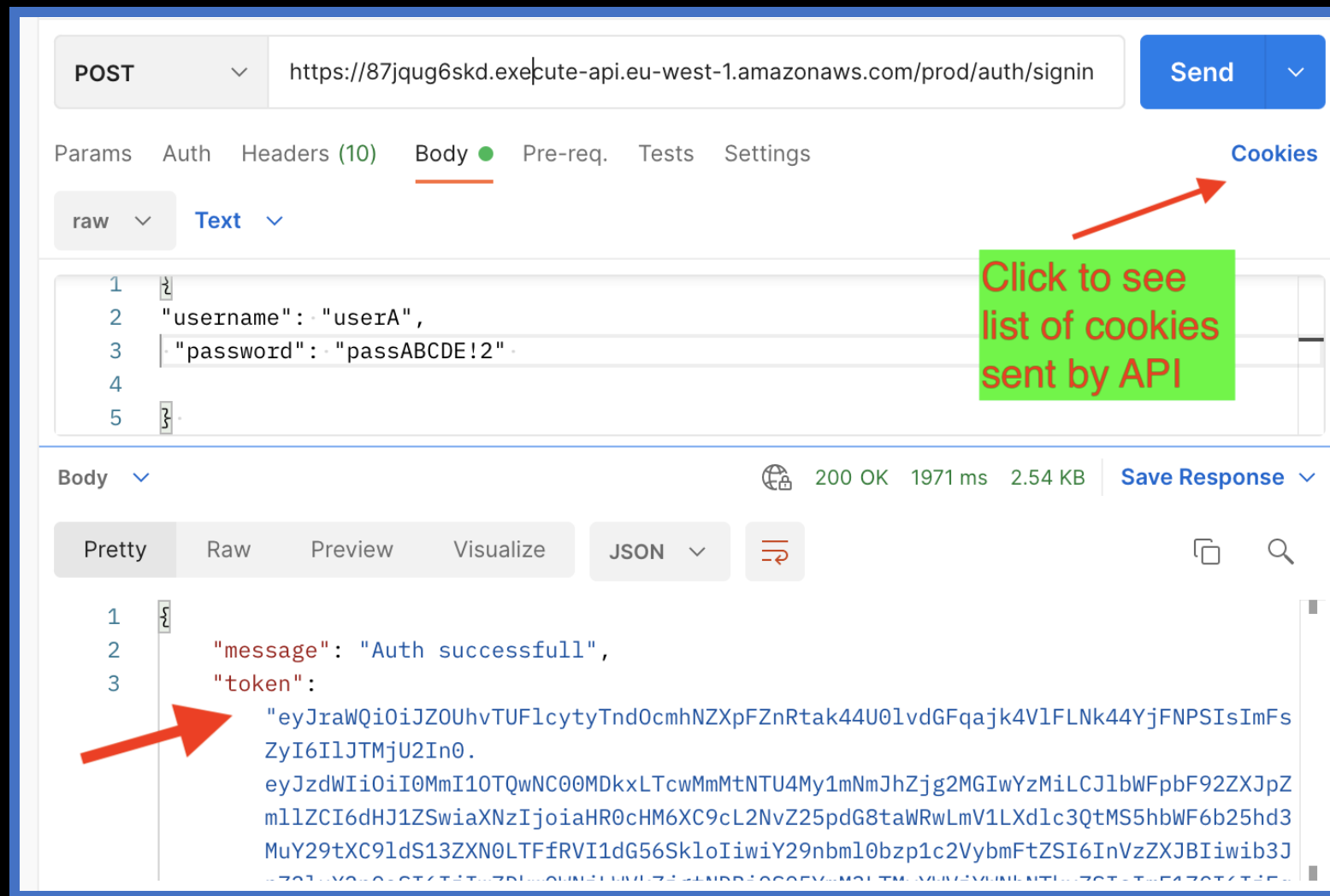
```
186
187 const client = new CognitoIdentityProviderClient({ region: "eu-west-1" });
188
189 export const handler: APIGatewayProxyHandlerV2 = async (event) => {
190   const { username, password } = JSON.parse(event.body);
191   const params: InitiateAuthCommandInput = {
192     ClientId: process.env.CLIENT_ID!,
193     AuthFlow: "USER_PASSWORD_AUTH",
194     AuthParameters: {
195       USERNAME: username,
196       PASSWORD: password,
197     },
198   };
199   try {
200     const command = new InitiateAuthCommand(params);
201     const { AuthenticationResult } = await client.send(command);
202     const token = AuthenticationResult.IdToken;
203
204     return {
205       statusCode: 200,
206       headers: {
207         "Access-Control-Allow-Headers": "*",
208         "Access-Control-Allow-Origin": "*",
209         "Set-Cookie": `token=${token}; SameSite=None; Secure; HttpOnly; Path=/; M
210       },
211       body: JSON.stringify({
212         message: "Auth successfull",
213         token: token,
214       }),
215     };
216   } catch (err) {.... }
217   };

```

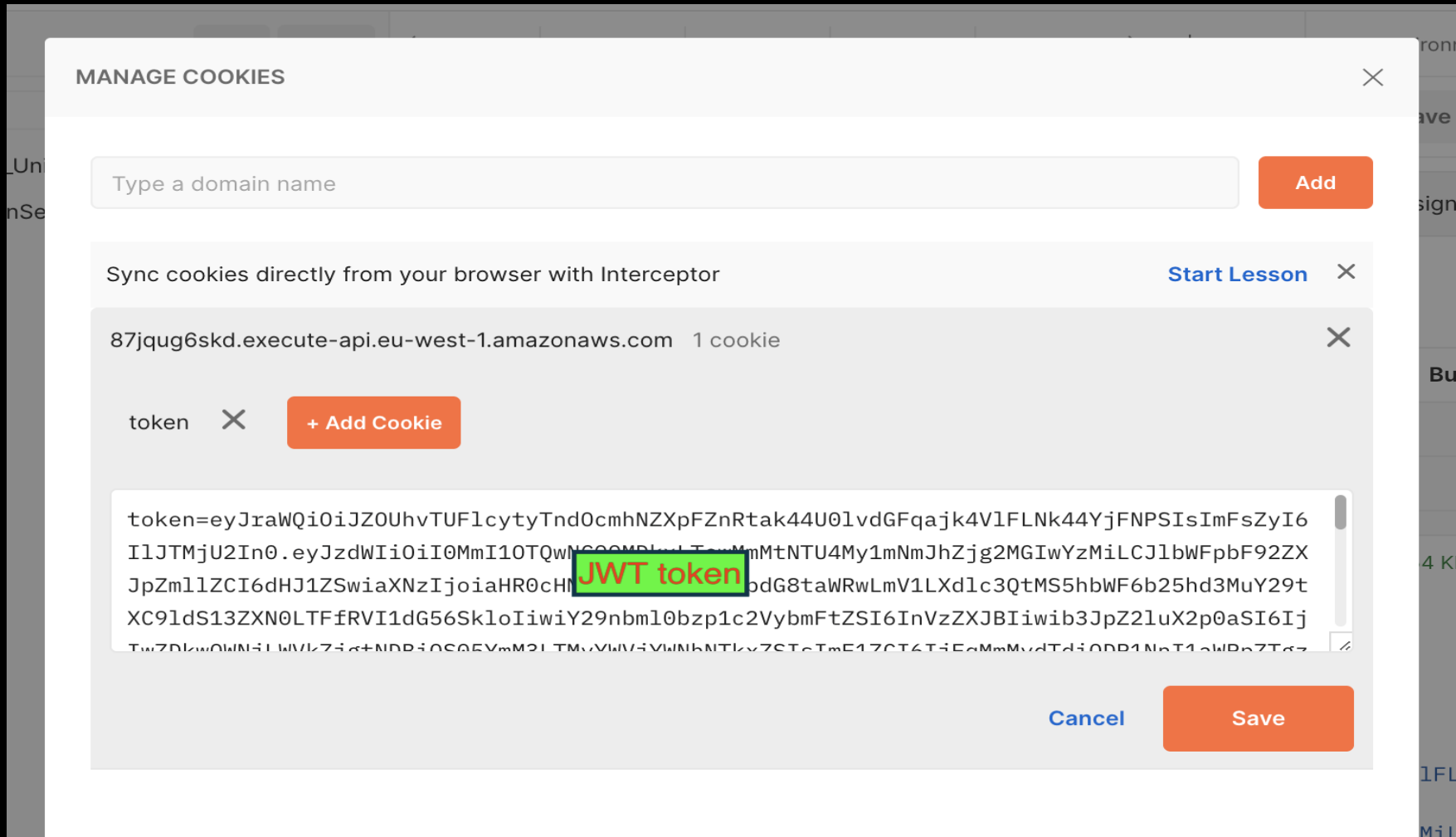
Restart Visual Studio Code to apply the latest changes

Update Now

# Demo – Sign In request / response

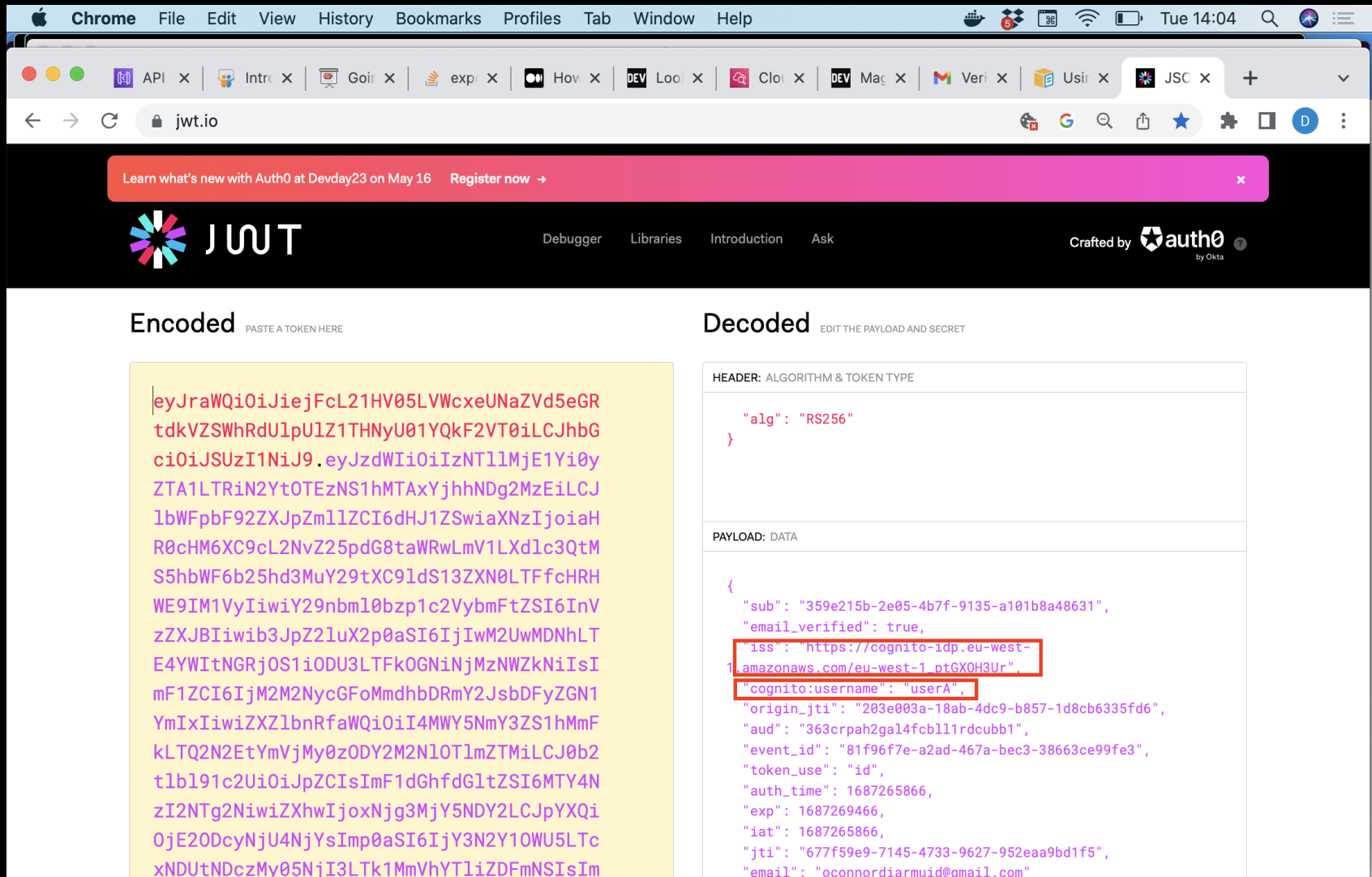


# Demo – Sign In JWT token

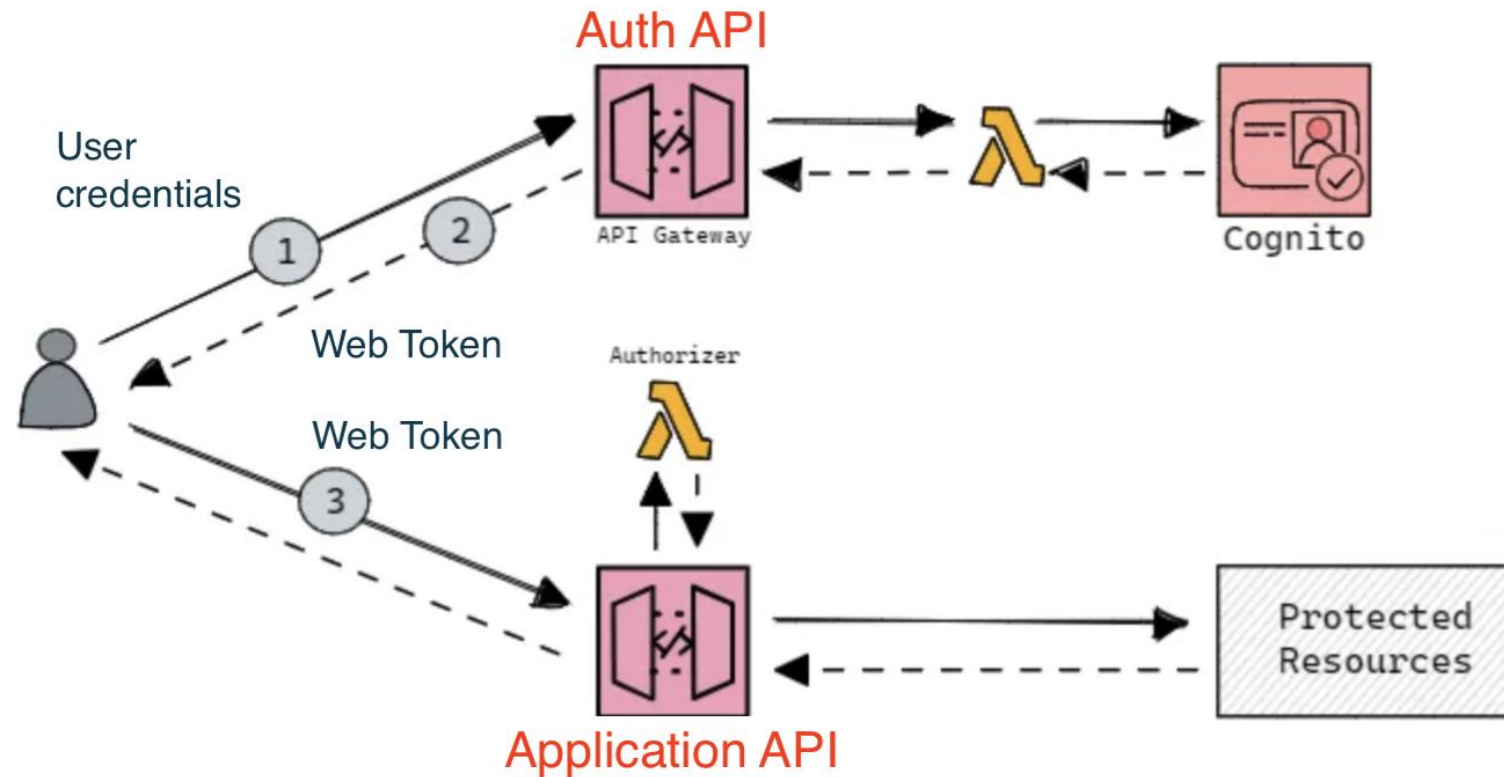




# JWT token decoding

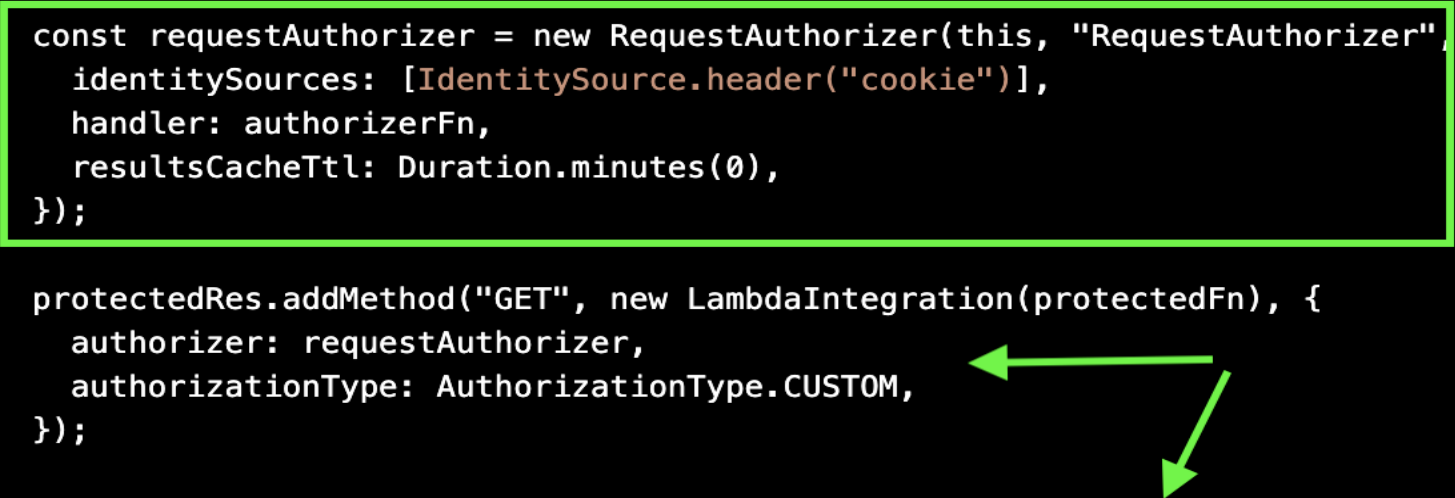


# Demo - Architecture



# Demo – App API infrastructure.

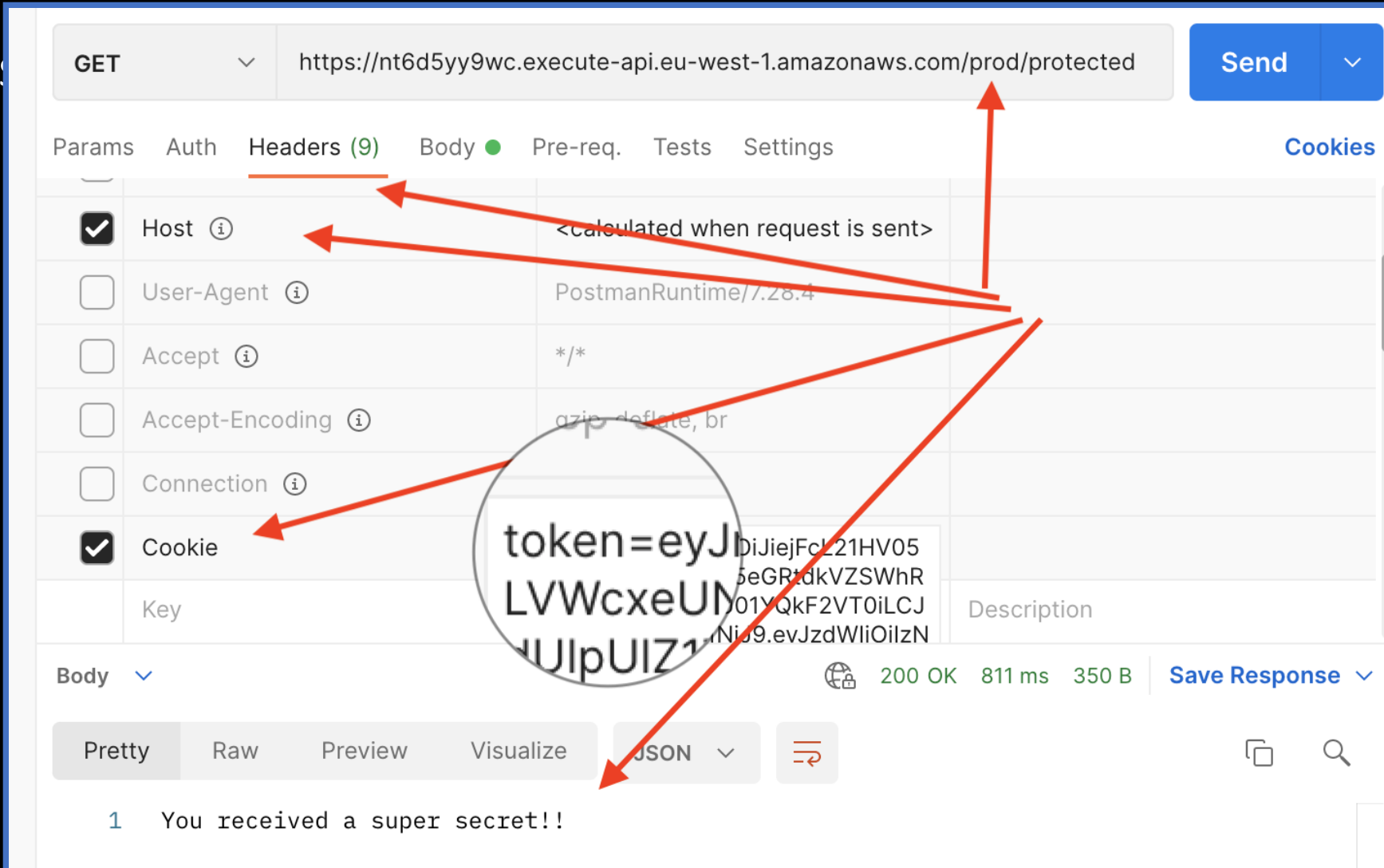
```
230 const api = new RestApi(this, "App Api", .....);
231
232 const protectedRes = api.root.addResource("protected");
233 const publicRes = api.root.addResource("public");
234
235 const protectedFn = new NodejsFunction(this, "ProtectedFn", ....);
236 const publicFn = new NodejsFunction(this, "PublicFn", ....);
237
238 const authorizerFn = new NodejsFunction(this, "AuthorizerFn", .....);
239
240 const requestAuthorizer = new RequestAuthorizer(this, "RequestAuthorizer", {
241     identitySources: [IdentitySource.header("cookie")],
242     handler: authorizerFn,
243     resultsCacheTtl: Duration.minutes(0),
244 });
245
246 protectedRes.addMethod("GET", new LambdaIntegration(protectedFn), {
247     authorizer: requestAuthorizer,
248     authorizationType: AuthorizationType.CUSTOM,
249 });
250
251 publicRes.addMethod("GET", new LambdaIntegration(publicFn));
252
```



The diagram consists of a green rectangular box highlighting lines 240 through 244 of the code. From the right side of this box, two green arrows originate. One arrow points horizontally to the left, ending at line 247, specifically at the 'authorizer' property in the 'protectedRes.addMethod' call. The second arrow points diagonally down and to the left, ending at line 249, specifically at the closing brace of the 'protectedRes.addMethod' call.

# Demo – Protected route HTTP request

- For AWS



# Demo – Path parameters

- Steps performed by authorizer:
  1. Parse HTTP Request Cookie header value & store in a local Map data structure.
  2. Find 'token' key in the Map.
  3. If not found:
    - Return an IAM policy Denying use of the App Web API.
  4. Verify the token - Decode and check user exists in the User pool.
  5. If successful verification:
    - Return IAM policy that Allows execution of the App Web API.
    - Else
    - Return policy Denying execution of the App Web API.

# API Keys & Usage plans

- API keys are alphanumeric string values that you distribute to application developers to grant access to your API.
  - Keys can be generated by the API Gateway service, or imported into API Gateway from an external source.
- A usage plan specifies who can access one or more deployed API stages (e.g. dev, prod) methods, and optionally sets the target request rate to start throttling requests.
  - A plan uses API keys to identify the clients, and determine their access rights to associated APIs.

# API Keys & Usage plans

