Module 6 - Enable fine-grained access control for your APIs

In this module, you will explore how to leverage Amazon Verified Permissions to secure access to Amazon API Gateway APIs using Amazon Cognito. Amazon Verified Permissions (AVP) streamlines the process of implementing fine-grained access control, reducing the time and complexity involved from weeks to just a few days. By managing and evaluating granular security policies that reference user attributes and groups, AVP enables you to ensure that only authorized users, based on their roles, have access to your APIs.

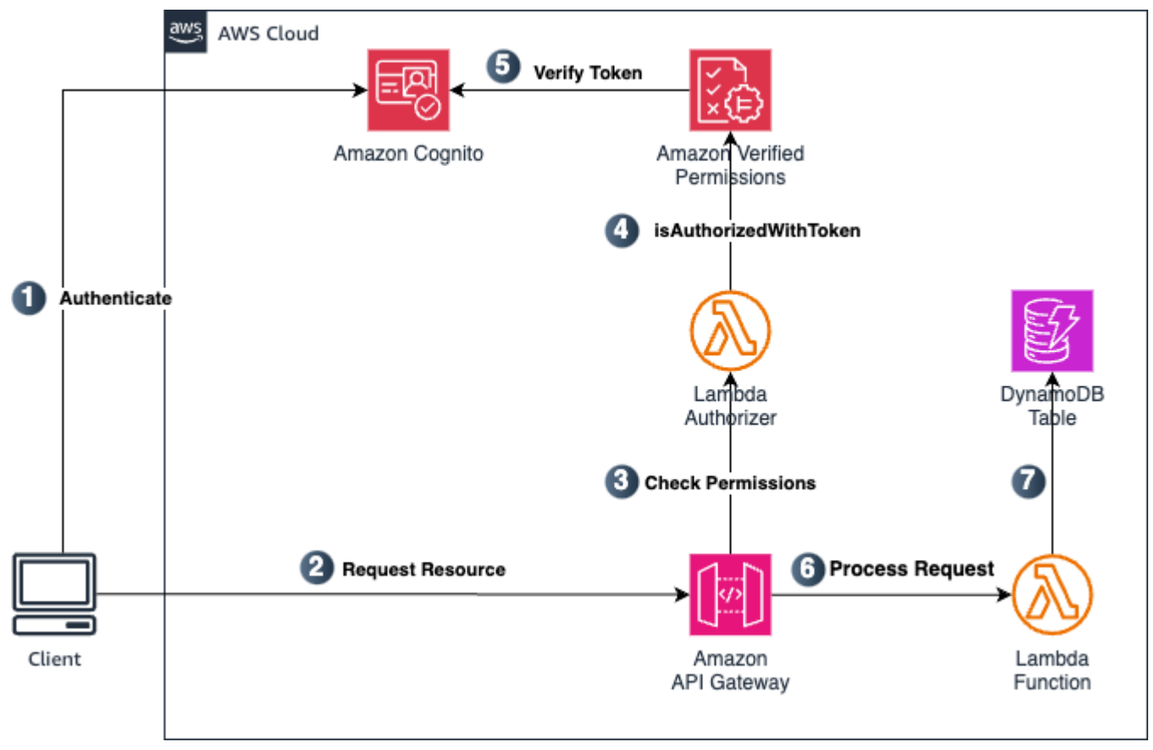
Role-Based Access Control (RBAC) is essential in cloud applications, providing a scalable and manageable way to ensure that users only have access to the resources necessary for their roles, thus enhancing overall security. Amazon Verified Permissions elevates this concept by introducing a more granular and condition-based access control mechanism. In this module, you will see how AVP simplifies the creation of complex authorization systems compared to traditional approaches. You'll implement an authorization example that integrates Amazon Verified Permissions with Amazon API Gateway.

Technologies you will work with:

- **Amazon Cognito:** Identity provider for managing user authentication.
- **Amazon API Gateway:** Fully managed service to create, publish, and secure APIs.
- **AWS Lambda:** Serverless compute service that acts as the backend application.
- **Amazon DynamoDB:** Scalable NoSQL database service for data storage.
- **Amazon Verified Permissions:** Fine-grained authorization service to manage and enforce access control policies.

Solution overview

The diagram below illustrates the architecture of the solution you will build during this workshop.



Let's review the solution:

1. **Authenticate:** The client authenticates with the Amazon Cognito user pool to obtain an authorization token.
2. **Request Resource:** The client sends a request to the Amazon API Gateway, including the Cognito token in the Authorization header.
3. **Check Permissions:** The API Gateway invokes a Lambda authorizer to check the permissions. The Lambda authorizer prepares an authorization query for Amazon Verified Permissions.
4. **isAuthorizedWithToken:** The Lambda authorizer sends the authorization query to Amazon Verified Permissions, which uses the token to validate the user's identity and assess the query.
5. **Verify Token:** Amazon Verified Permissions checks the token against the Amazon Cognito user pool to ensure its validity and evaluates the authorization query against the policies stored in the policy store.
6. **Process Request:** Amazon Verified Permissions returns an authorization decision to the Lambda authorizer. Based on this decision, the API Gateway either allows or denies the API request. If allowed, the API Gateway invokes the backend Lambda function.
7. **Database Interaction:** The backend Lambda function interacts with the DynamoDB table to perform the necessary read or write operations.

Learning objectives

By the end of this module, you will be able to:

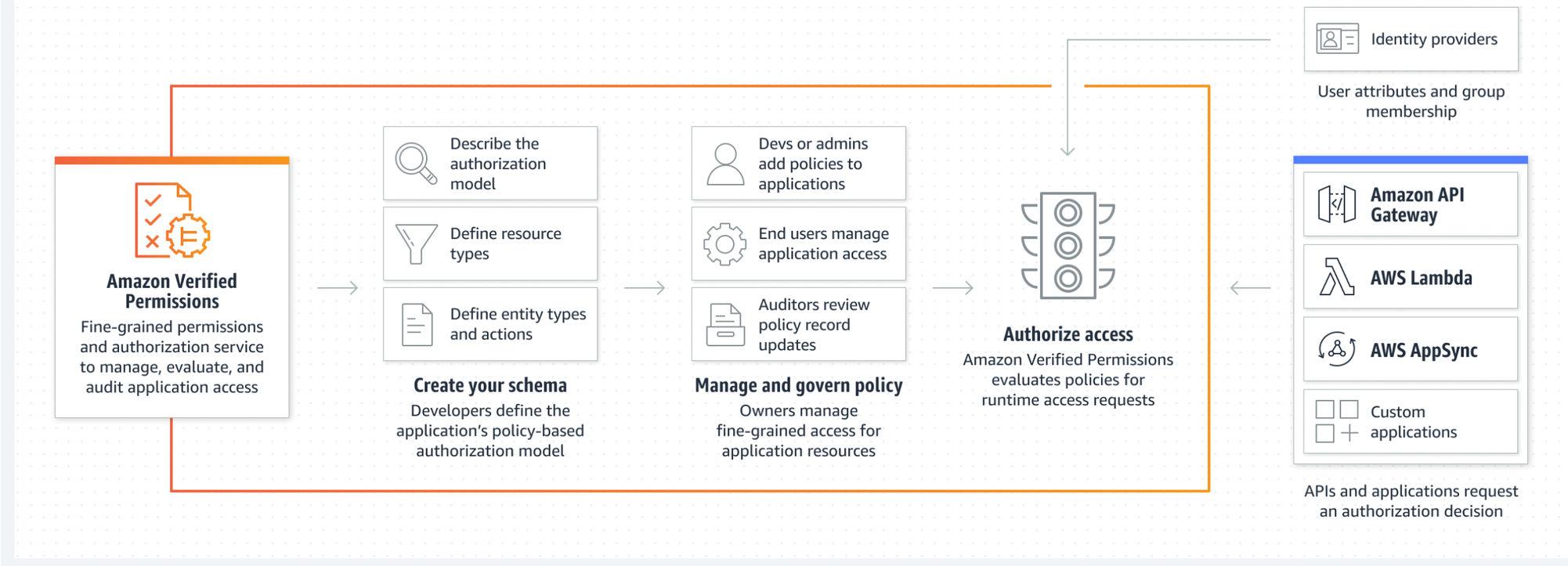
- Quickly start using Amazon Verified Permissions to authorize backend requests.
- Manage users and their roles using Amazon Cognito groups.
- Implement custom permissions management in Amazon API Gateway using AWS Lambda authorizers with Amazon Verified Permissions.

Additionally, you will review a practical example of how Amazon Verified Permissions can be used to secure requests to a backend application.

Estimated Duration: 60 min

Introduction to Amazon Verified Permissions

Amazon Verified Permissions is a fully managed authorization service that uses the provably correct Cedar policy language, so you can build more secure applications. With Verified Permissions, developers can build applications faster by externalizing authorization and centralizing policy management. They can also align authorization within the application with Zero Trust principles. Security and audit teams can better analyze and audit who has access to what within applications.



The use cases for Amazon Verified Permissions include:

- **Define a fine-grained authorization model** - Create policies from templates and enforce those controls in Amazon API Gateway and AWS AppSync.
- **Grant fine-grained permissions within applications** - Administrators can create application-wide policies, and developers can grant user permissions to access data and resources.
- **Audit permissions across applications** - Review policy model changes and monitor authorization requests using Verified Permissions.
- **Centralize the policy administration system** - Create and centrally store policy-based access controls, and meet your application latency requirements with millisecond processing.

In this module, you will take a deep dive into Amazon Verified Permissions. We will explore how to implement an effective authorization solution tailored for a sample Real Estate application, showcasing the integration of AVP with Amazon API Gateway and other AWS services.

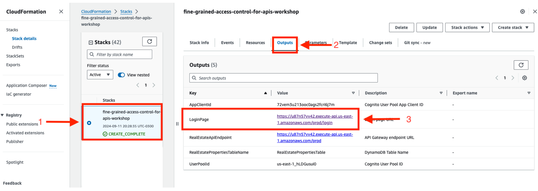
Authenticating with Amazon Cognito

The Real Estate application uses an Amazon Cognito user pool to authenticate users. The Amazon Cognito user pool OAuth 2.0 authorization server issues tokens in response to the authorization code grant. In response to your successful authentication request, the authorization server appends an authorization code in a code parameter to your callback URL. You must then exchange the code for ID, access, and refresh tokens with the token endpoint. The authorization code grant is the most secure form of authorization grant. It doesn't show token contents directly to your users. Instead, your app is responsible for retrieving and securely storing your user's tokens.

Using the Amazon Cognito Hosted UI

In this section we will explore the **Amazon Cognito Hosted UI** and how we can use it to register new users and to sign in existing users.

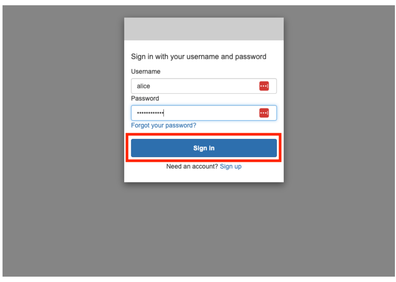
The login page address is available in the CloudFormation console Outputs tab of your workshop stack:



In fact, this is the invoke URL of your Amazon API Gateway stage with the `/login` path appended to it. This page will redirect you to the **Amazon Cognito Hosted UI**.

There are 3 previews created users: `alice`, `bob` and `charlie`. All were setted as a default password: `Password123!`.

Sign in with one of the users:



You should be redirected back to your application and see the id and access tokens on the screen:

The tokens we see here are JWT Tokens and they are a topic of their own, but while we are here lets take a peek at their structure and the information they hold.

Note the **Access Token**.

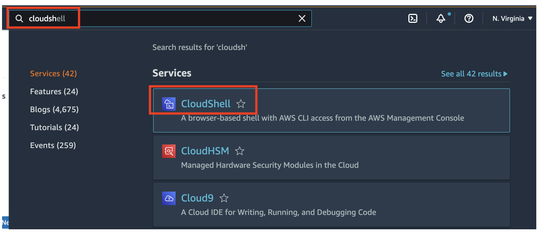
Retrieve the tokens for `alice`, `bob` and `charlie` and copy to a text editor (you will need the tokens later).

Access AWS CloudShell

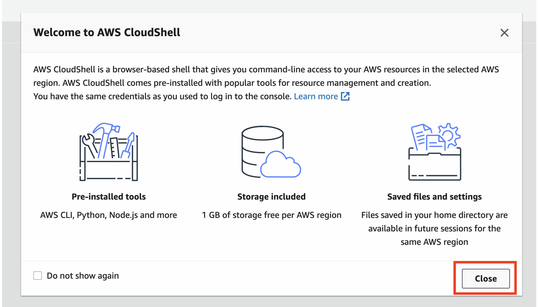
Now that the user pool is configured, you will use a service called AWS CloudShell and leverage the command lines to test calls via cli.

Tip
AWS CloudShell is a browser-based shell terminal that makes it easy to securely manage, explore, and interact with your AWS resources to run the API command.

1. In the Search Bar of the AWS Management Console type CloudShell, and chose CloudShell from the search options:



2. Choose *Close*, to progress past the welcome alert:



Export Tokens as Environment Variables

Paste the `"access_token"` for each user and assign it to an environment variable:

```
export RE_ACCESS_TOKEN_ALICE=<access_token_content>
export RE_ACCESS_TOKEN_BOB=<access_token_content>
export RE_ACCESS_TOKEN_CHARLIE=<access_token_content>
```

Feel free to save the tokens in the same ws-env.sh file for later use:

```
env | grep "RE_ | sed -e 's/^/export /' > ws-env.sh
```

We can check the content of `"access_token"`:

```
jq -R 'split(".") | .[1] | @base64d | fromjson' <<< "$RE_ACCESS_TOKEN_ALICE"
```

The output should look something like this:

```
[cloudshell-user@ip-10-134-36-55 ~]$ jq -R 'split(".") | .[1] | @base64d | fromjson' <<< "$RE_ACCESS_TOKEN_ALICE"
{
  "sub": "04d8c4b8-00c1-7064-6404-121fa909d0d6",
  "iss": "https://cognito-idp.us-east-1.amazonaws.com/us-east-1_hLDGusU8",
  "version": 2,
  "client_id": "72vem3uj13oocBagv2fcri6j7m",
  "origin_jti": "a369fc9a-9ca2-4288-a74b-0ef77c6ea0b0",
  "token_use": "access",
  "scope": "openid profile RealEstateResourceServer/RealStateApi email",
  "auth_time": 1726262859,
  "exp": 1726349259,
  "iat": 1726262859,
  "jti": "10750a4e-dc69-4bb9-9afb-b9225b947f83",
  "username": "alice"
}
```

Use Tokens to Access APIs

1. Get the RealEstateApiEndpoint in the Cloudformation Stack Output, and export it as an environment variable.

```
export RE_API_ENDPOINT=<RealStateApiEndpoint>
```

You can now use the RE_ACCESS_TOKEN_ALICE in the Authorization header when making requests to your API endpoints. For example:

```
curl -X GET $RE_API_ENDPOINT/real-estate-properties \
-H "Authorization: Bearer $RE_ACCESS_TOKEN_ALICE" | jq
```

You can test the tokens for `bob` and `charlie` as well

This allows you to authenticate your API calls and manage access securely based on the user roles configured in your Cognito User Pool.

Conclusion

In this chapter we have explored the **Amazon Cognito Hosted UI** and how we can use it to sign in existing users. We have also taken a peek at the tokens that are returned by the **Amazon Cognito Hosted UI**. It helps ensure that your API endpoints are accessed only by authenticated users, as defined by your Cognito

Exploring the Real Estate API

The Real Estate API is a simple RESTful API that allows you to manage real estate properties. The API is secured using Amazon Cognito user pools and Amazon API Gateway. Amazon Cognito authorizer is used to authenticate users and authorize access to the API. The client must first sign the user in to the user pool, obtain an identity or access token, and then call the API method with one of the tokens, which are typically set to the request's Authorization header. The API call succeeds only if the **required token is supplied** and the **supplied token is valid**, otherwise, the client isn't authorized to make the call because the client did not have credentials that could be authorized. No additional authorization checks are performed by the API Gateway at this time.

The application is pre-deployed using AWS CloudFormation.

Below is the list of resources created in the API Gateway:

- **GET /real-estate-properties** - Returns a list of all real estate properties.
- **POST /real-estate-properties** - Adds a new real estate property.
- **GET /real-estate-properties/{id}** - Returns a real estate property by ID.
- **PUT /real-estate-properties/{id}** - Updates a real estate property by ID.
- **DELETE /real-estate-properties/{id}** - Deletes a real estate property by ID.
- **GET /login** - Redirects to the Amazon Cognito Hosted UI for user sign-in.
- **GET /logout** - Redirects to the Amazon Cognito Hosted UI for user sign-out.

Working with the Real Estate API

Getting the list of real estate properties

To get a list of real estate properties, use the following API call:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_ALICE" | jq
```

The response looks like this:

```
[
  {
    "location": "Malibu, California",
    "description": "A beautiful beachfront villa with a stunning ocean view.",
    "id": "1bbe07a8-bab9-47e1-a7ac-668f3dd394f8",
    "price": 950000,
    "name": "Beachfront Villa"
  },
  {
    "location": "Aspen, Colorado",
    "description": "A cozy cabin in the mountains, perfect for a winter getaway.",
    "id": "2cde08a9-cdc9-48a8-bb8a-74b9e9a748ac",
    "price": 350000,
    "name": "Mountain Cabin"
  }
]
```

Adding a new real estate property

To add a new real estate property, use the following API call:

```
curl -X POST "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_ALICE" \
  -H "Content-Type: application/json" \
  -d '{
      "name": "Luxury Apartment",
      "description": "A modern luxury apartment in the city center.",
      "price": 1200000,
      "location": "New York, New York"
    }' | jq
```

The response looks like this:

```
{
  "message": "Real estate property created successfully",
  "id": "52a61a11-8e72-43f4-9430-67fe126dfa27"
}
```

You can verify that the real estate property was added by getting the list of real estate properties again.

Getting a real estate property by ID

To get a real estate property by ID, use the following API call:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties/<generated-property-id>" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_ALICE" | jq
```

The response looks like this:

```
{
  "location": "New York, New York",
  "description": "A modern luxury apartment in the city center.",
  "id": "52a61a11-8e72-43f4-9430-67fe126dfa27",
  "price": 1200000,
  "name": "Luxury Apartment"
}
```

Updating a real estate property by ID

To update a real estate property by ID, use the following API call:

```
curl -X PUT "$RE_API_ENDPOINT/real-estate-properties/generated-property-id" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_ALICE" \
  -H "Content-Type: application/json" \
  -d '{
      "name": "Updated Luxury Apartment",
      "description": "An updated description for the luxury apartment.",
      "price": 1300000,
      "location": "New York, New York"
    }' | jq
```

The response looks like this:

```
{
  "message": "Real Estate Property updated successfully"
}
```

You can verify that the real estate property was updated by getting the real estate property by ID again.

Deleting a real estate property by ID

To delete a real estate property by ID, use the following API call:

```
curl -X DELETE "$RE_API_ENDPOINT/real-estate-properties/generated-property-id" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_ALICE" | jq
```

The response looks like this:

```
{
  "message": "Real Estate Property deleted successfully"
}
```

You can verify that the real estate property was deleted by getting the real estate property by ID again. The response should be an error message.

```
{
  "message": "Real Estate Property not found"
}
```

Authorization Concerns

If you try to access the API without a token, you will get an error message:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties" | jq
```

The response looks like this:

```
{
  "message": "Unauthorized"
}
```

If you try to access the API with an invalid token, you will get the same error message:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: invalid_token" | jq
```

That happens because the API Gateway is configured to use the Amazon Cognito user pool authorizer, which validates the token.

However, if you try to access the API with a valid token belonging to a different user, you will not have any issues:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_BOB" | jq
```

The reason is that the API Gateway only checks if the token is valid and does not perform any additional authorization checks.

Summary

In this section, you learned how to work with the Real Estate application API. You learned how to get a list of real estate properties, add a new real estate property, get a real estate property by ID, update a real estate property by ID, and delete a real estate property by ID. You also learned about the authorization concerns when working with the API.

Creating Cognito groups

In this section, you will learn how to create groups in Amazon Cognito user pools. You will create two groups: `Admin` and `User`. The `Admin` group will have permissions to perform all operations on the Real Estate API, while the `User` group will have permissions to perform only read operations.

Creating the Admin group

1. Open the Amazon Cognito console. Make sure you are in the same region where you are running the workshop.
2. Choose the **RealEstateUserPool** user pool created by the CloudFormation stack. If in doubt, check the stack outputs in the CloudFormation console to find the user pool ID.
3. Navigate to the **Groups** tab.
4. Choose **Create group**.
5. Enter `Admin` as the group name, leave the other options empty and click in **Create group**.
6. Click the group you just created.
7. Click in the **Add users to group** button.
8. Select the users you want to add to the group and choose **Add**. E.g., let's make alice an admin.

Creating the User group

1. Get back to the **Groups** tab and repeat the steps above to create a new group named `User` .
2. Add `bob` to the `User` group.

Summary

In this section, you created two groups in Amazon Cognito user pools: `Admin` and `User`. You added `alice` to the `Admin` group and `bob` to the `User` group. In the next section, you will learn how to configure the API Gateway to authorize users based on their group membership.

Configuring Amazon Verified Permissions

In this section, you will learn how to leverage Amazon Verified Permissions to manage and evaluate granular security policies that reference user attributes and groups. With a few clicks, you will enforce that only users in authorized Amazon Cognito groups have access to the application's APIs.

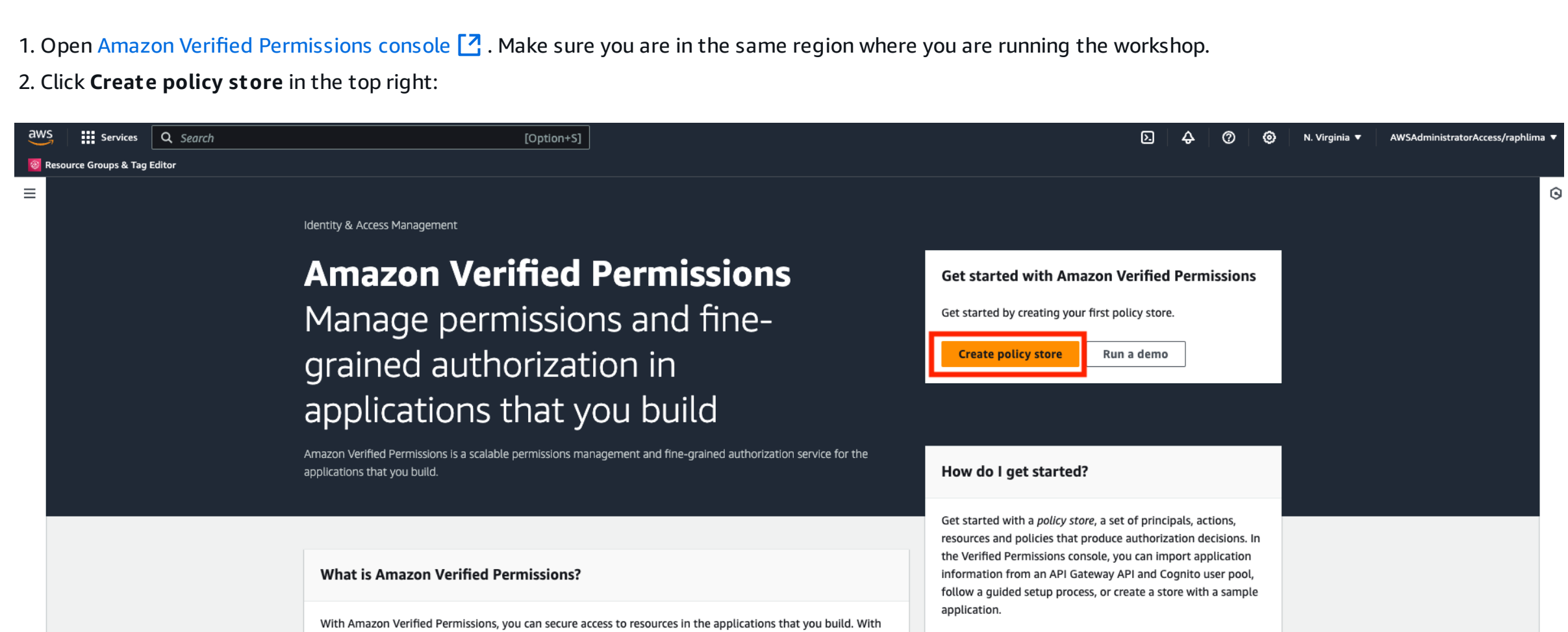
Creating a policy store

Policy store is a logical container that describes your authorization model. It includes:

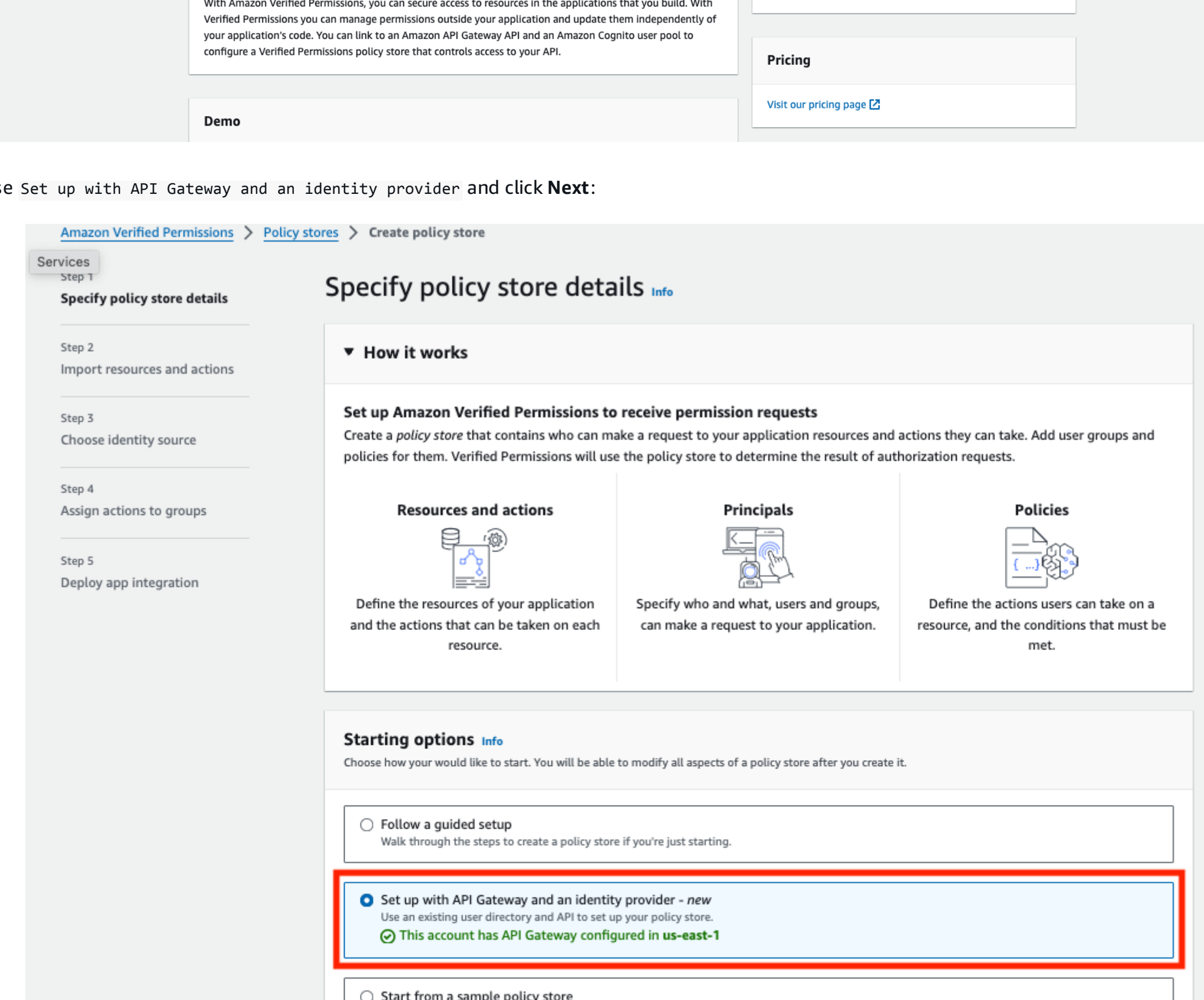
- Definition of resource types that you are controlling access to
- Definition of principal types that are performing actions on your resources
- List of actions that principals can take on resources
- Policies that define authorization rules

The policy store is a container for your policies. Your application will authorize request against a specific policy store.

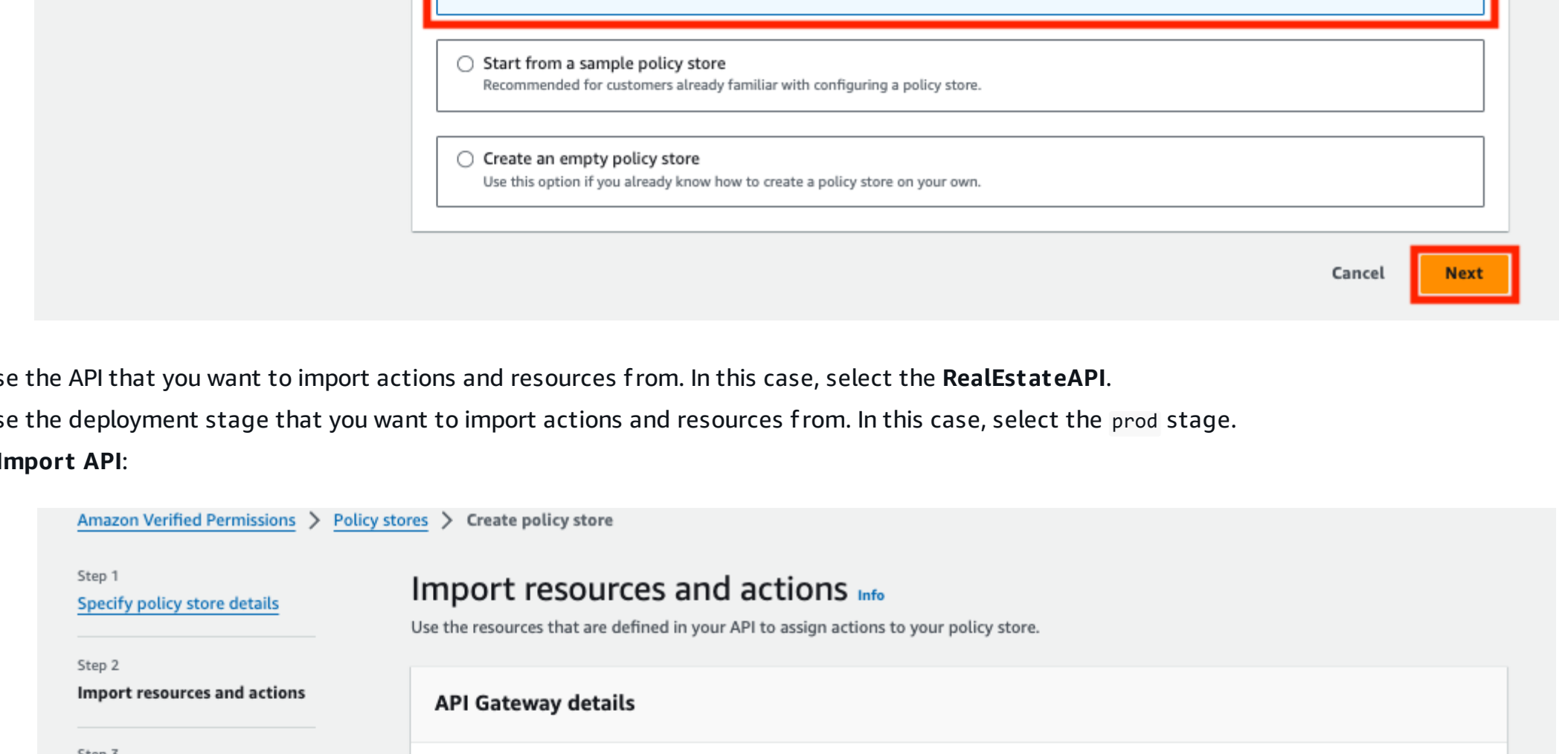
1. Open Amazon Verified Permissions console. Make sure you are in the same region where you are running the workshop.
2. Click **Create policy store** in the top right:



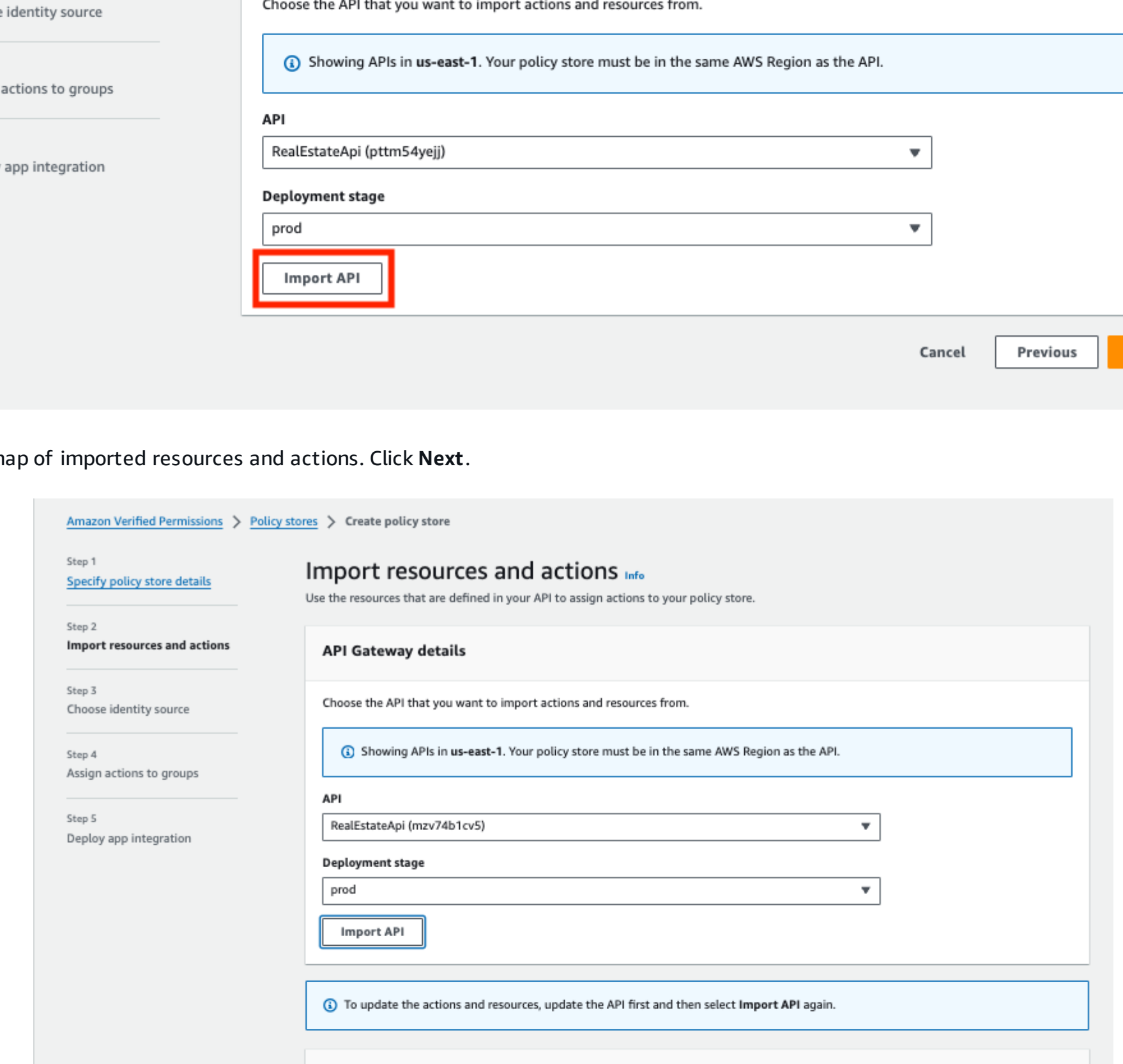
3. Choose `Set up with API Gateway and an identity provider` and click **Next**:



4. Choose the API that you want to import actions and resources from. In this case, select the **RealEstateAPI**.
5. Choose the deployment stage that you want to import actions and resources from. In this case, select the `prod` stage.
6. Click **Import API**:

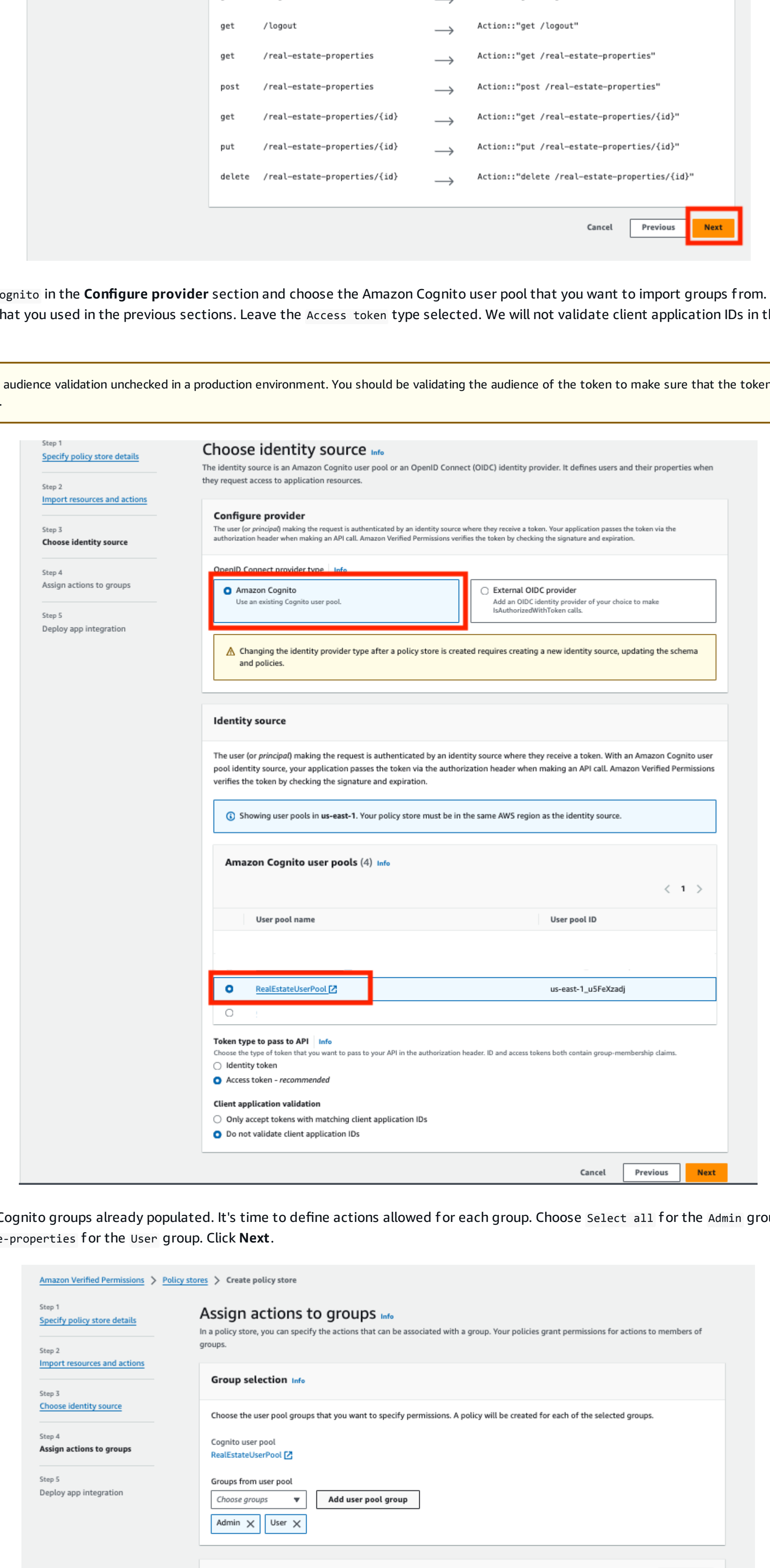


7. You will see the map of imported resources and actions. Click **Next**.

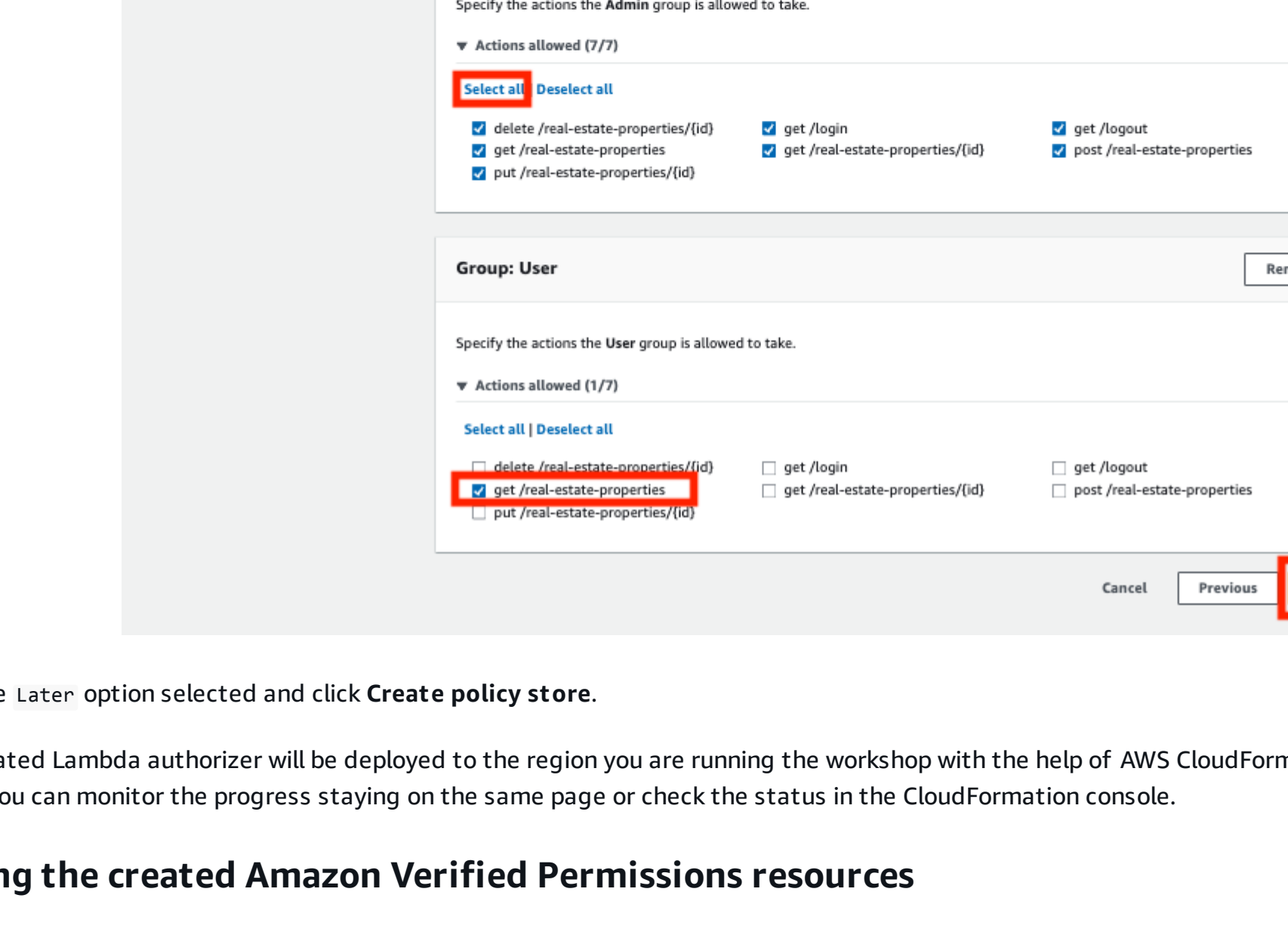


8. Choose `Amazon Cognito` in the **Configure provider** section and choose the Amazon Cognito user pool that you want to import groups from. In this case, it's the same user pool that you used in the previous sections. Leave the `Access token` type selected. We will not validate client application IDs in this workshop. Click **Next**.

> ⚠ Never leave the audience validation unchecked in a production environment. You should be validating the audience of the token to make sure that the token is intended for your application.



9. You will see the Cognito groups already populated. It's time to define actions allowed for each group. Choose `Select all` for the `Admin` group and choose only `get /real-estate-properties` for the `User` group. Click **Next**.



10. Leave the `Later` option selected and click **Create policy store**.

A newly created Lambda authorizer will be deployed to the region you are running the workshop with the help of AWS CloudFormation. This requires some time to complete. You can monitor the progress staying on the same page or check the status in the CloudFormation console.

Exploring the created Amazon Verified Permissions resources

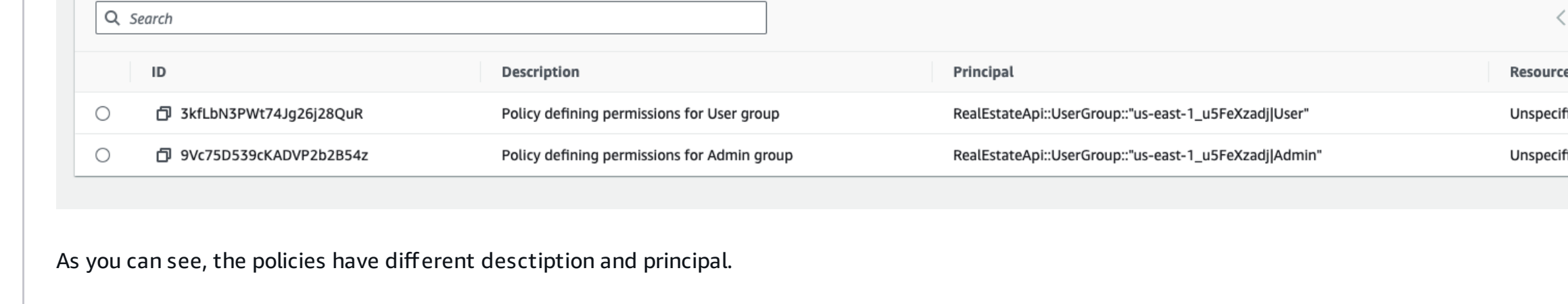
While the Lambda authorizer is being deployed, let's explore the resources created in the Amazon Verified Permissions console.

Policies

Policies define authorization rules. They are evaluated against the request context to determine if the request is authorized. You can define multiple policies in a policy store.

Let's explore the policies created for the RealEstateAPI.

Navigate to the **Policies** tab in the Amazon Verified Permissions console. You will see two policies created for the RealEstateAPI:



As you can see, the policies have different desctiption and principal.

The policy for the `Admin` group allows all actions on the RealEstateAPI:

```
permit(
  principal in RealEstateApi::UserGroup::"us-east-1_u5FeXzadj|Admin",
  action in [ RealEstateApi::Action::"delete /real-estate-properties/{id}", RealEstateApi::Action::"get /login", RealEstateApi::Action::"get /logout", RealEstateApi
  resource
);
```

The policy for the `User` group allows only the `get /real-estate-properties` action:

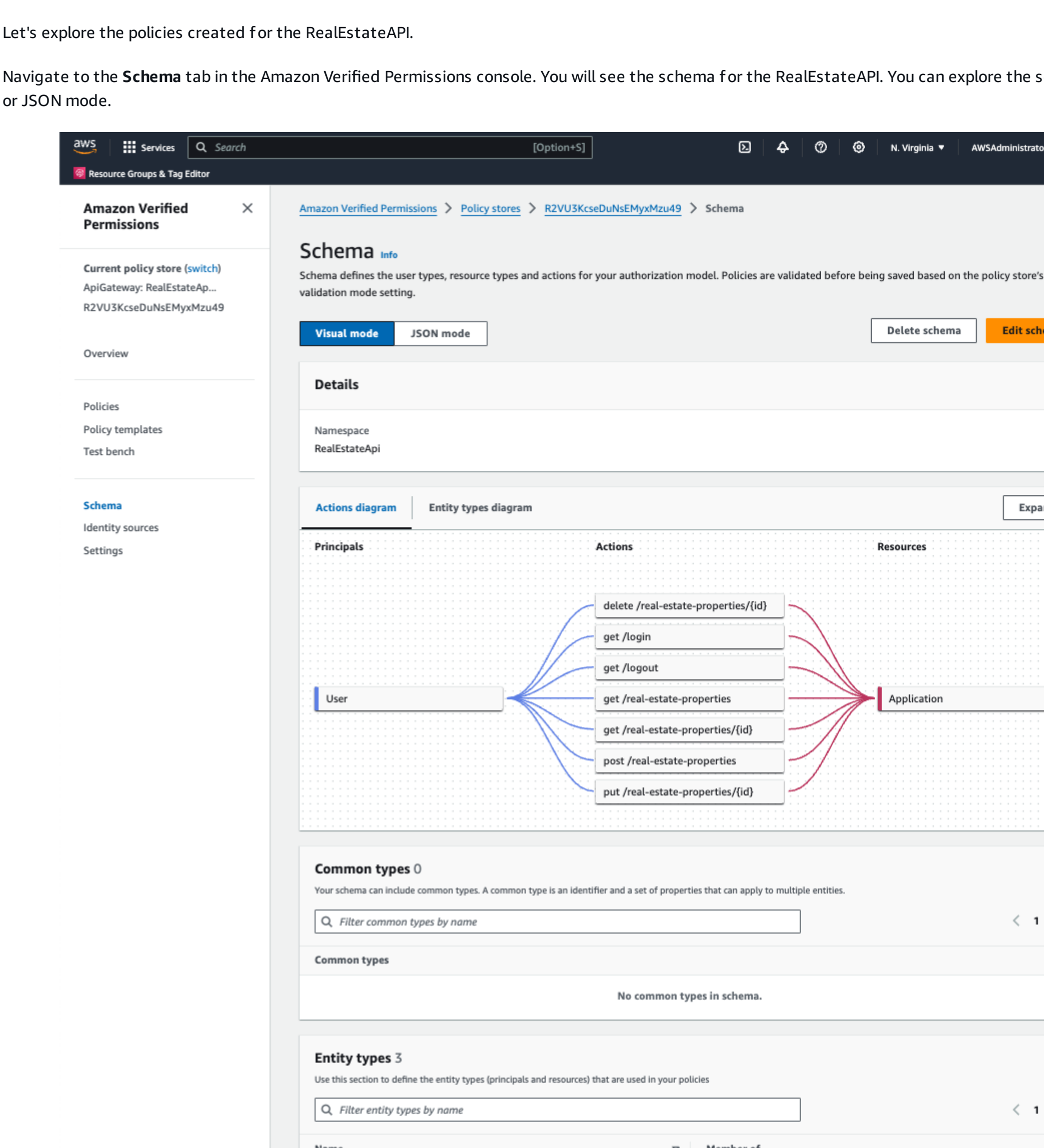
```
permit(
  principal in RealEstateApi::UserGroup::"us-east-1_u5FeXzadj|User",
  action in [ RealEstateApi::Action::"get /real-estate-properties" ],
  resource
);
```

Schema

As with all code, it is possible to make mistakes in Cedar policies that result in the code not behaving as expected. To validate a policy, Cedar needs information about the system. It needs to know the correct names of Entity Types, the attributes they possess, the allowed parent/child relationships, and whether the entities can act as principals or resources (or both). All of this is provided to Cedar through a schema file.

Let's explore the policies created for the RealEstateAPI.

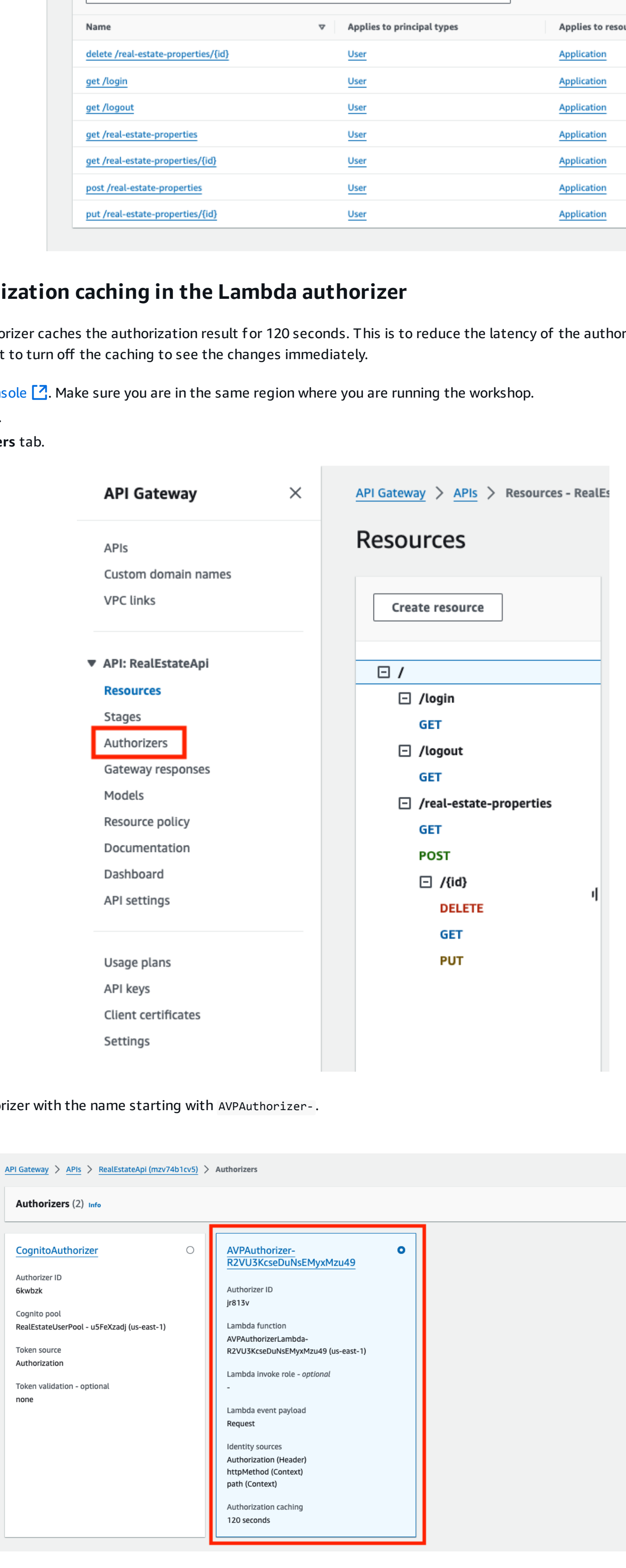
Navigate to the **Schema** tab in the Amazon Verified Permissions console. You will see the schema for the RealEstateAPI. You can explore the schema in the Visual or JSON mode.



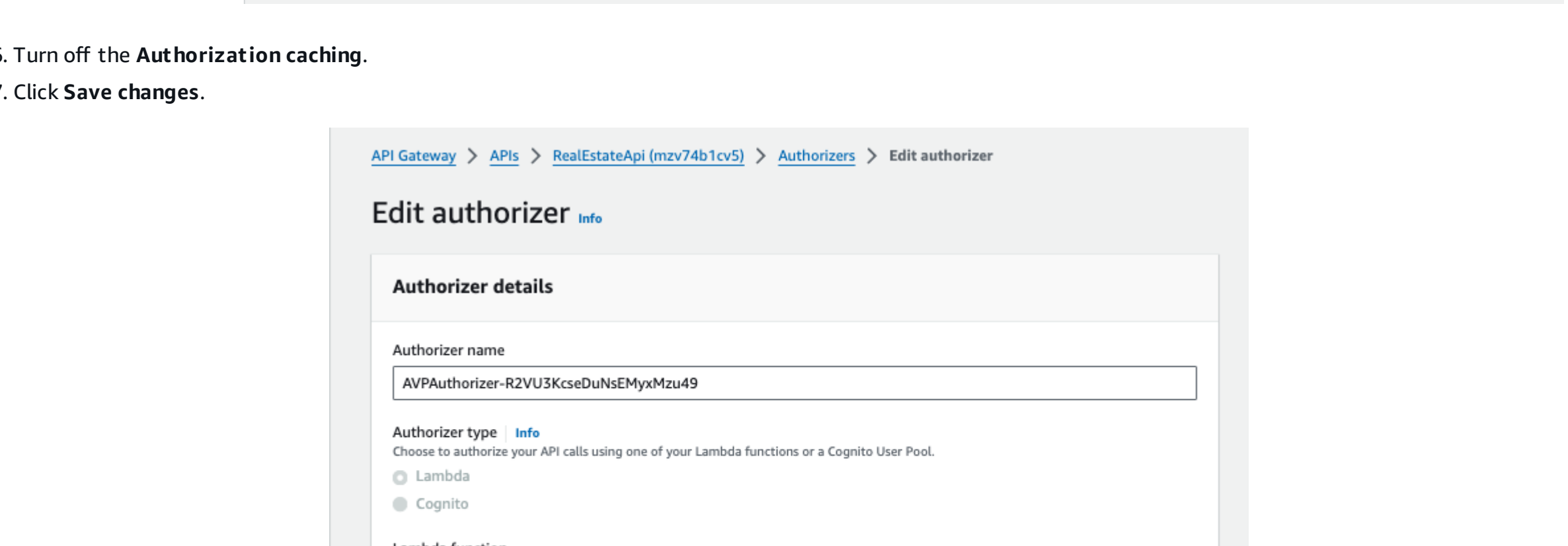
Turning off authorization caching in the Lambda authorizer

By default, the Lambda authorizer caches the authorization result for 120 seconds. This is to reduce the latency of the authorization process. However, during development, you might want to turn off the caching to see the changes immediately.

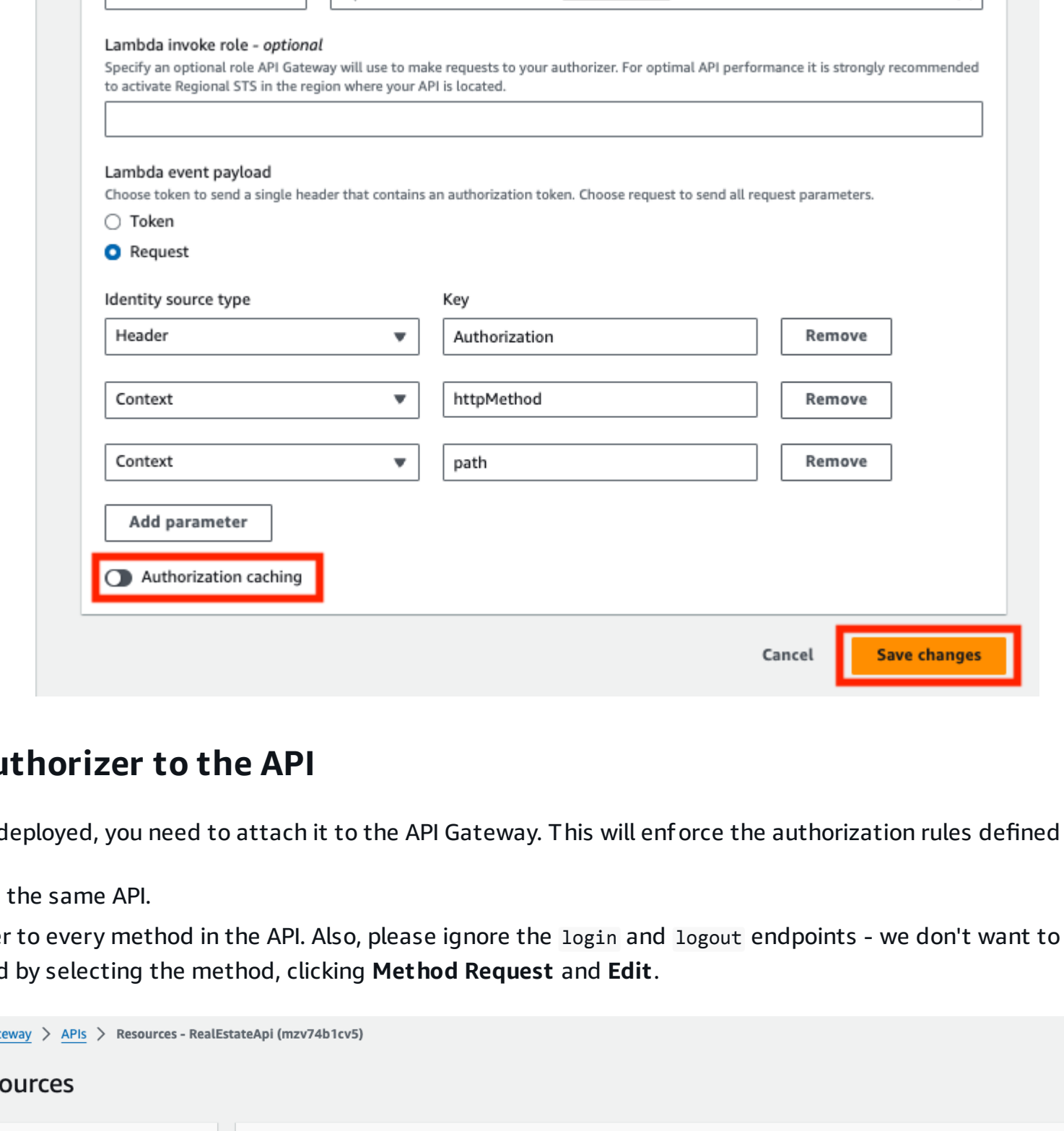
1. Open the API Gateway console. Make sure you are in the same region where you are running the workshop.
2. Choose the `RealEstateApi`.
3. Navigate to the **Authorizers** tab.



4. Click on the Lambda authorizer with the name starting with `AVPAuthorizer-`.
5. Click **Edit**.



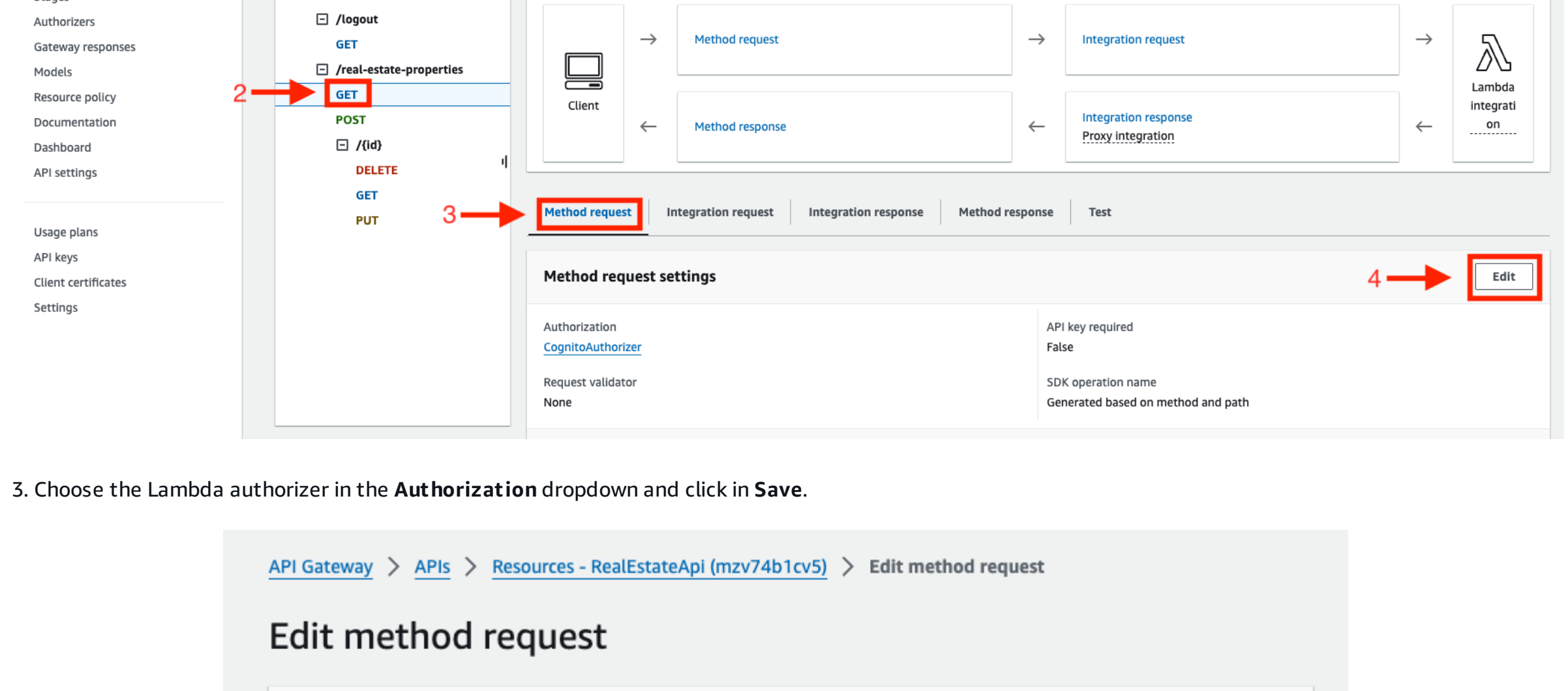
6. Turn off the **Authorization caching**.
7. Click **Save changes**.



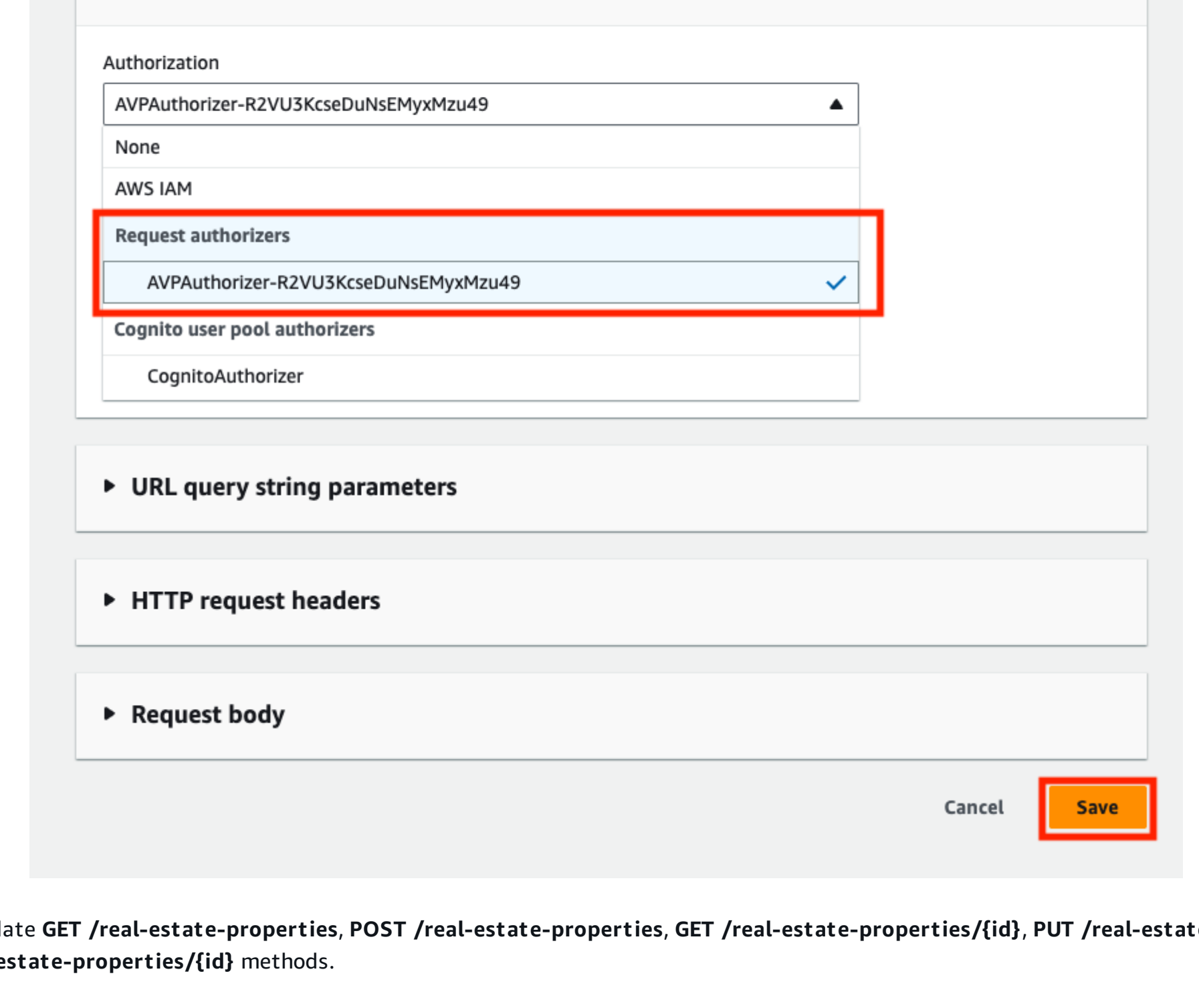
Attaching Lambda authorizer to the API

Once the Lambda authorizer is deployed, you need to attach it to the API Gateway. This will enforce the authorization rules defined in the policy store.

1. First click in Resources tab at the same API.
2. Attach the Lambda authorizer to every method in the API. Also, please ignore the `login` and `logout` endpoints - we don't want to authorize these endpoints. You can update every method by selecting the method, clicking **Method Request** and **Edit**.



3. Choose the Lambda authorizer in the **Authorization** dropdown and click in **Save**.



Don't forget to update **GET /real-estate-properties**, **POST /real-estate-properties**, **GET /real-estate-properties/{id}**, **PUT /real-estate-properties/{id}**, and **DELETE /real-estate-properties/{id}** methods.

Once done, click **Deploy API**, choose the prod stage, and click **Deploy**.

The API Gateway configuration is now complete. You might need to wait a few minutes for the changes to propagate.

Summary

In this section, you created a policy store in **Amazon Verified Permissions** to manage and evaluate granular security policies that reference user attributes and groups. You imported actions and resources from the RealEstateApi and groups from the Amazon Cognito user pool. You defined actions allowed for each group and created policies that define authorization rules. Finally you attached the Lambda authorizer to the API Gateway to enforce the authorization rules defined in

Reviewing Lambda authorizer

In this section, you will review the Lambda authorizer that was automatically created in the previous section. The Lambda authorizer is responsible for authorizing users before they can access the API. The Lambda authorizer is associated with the API Gateway methods that you configured in the previous section.

Reviewing the Lambda authorizer

1. Open the AWS Lambda console.
2. Filter the list of functions by the name of the Lambda authorizer function. It should contain `AVPAuthorizerLambda`.
3. Choose the Lambda authorizer function to open its details page.
4. Choose index.js to open the code editor.
5. Review the code of the Lambda authorizer function. The function is responsible for validating the incoming request and returning an IAM policy that allows or denies access to the API Gateway method.

Below is the code for the Lambda authorizer function:

```
const { VerifiedPermissions } = require('@aws-sdk/client-verifiedpermissions');
const policyStoreId = process.env.POLICY_STORE_ID;
const namespace = process.env.NAMESPACE;
const tokenType = process.env.TOKEN_TYPE;
const resourceType = `${namespace}::Application`;
const resourceId = namespace;
const actionType = `${namespace}::Action`;

const verifiedpermissions = !!process.env.ENDPOINT
  ? new VerifiedPermissions({
    endpoint: `https://${process.env.ENDPOINT}ford.${process.env.AWS_REGION}.amazonaws.com`,
  })
  : new VerifiedPermissions();

function getContextMap(event) {
  const hasPathParameters = Object.keys(event.pathParameters).length > 0;
  const hasQueryString = Object.keys(event.queryStringParameters).length > 0;
  if (!hasPathParameters && !hasQueryString) {
    return undefined;
  }
  const pathParametersObj = !hasPathParameters ? {} : {
    pathParameters: {
      // transform regular map into smithy format
      record: Object.keys(event.pathParameters).reduce((acc, pathParamKey) => {
        return {
          ...acc,
          [pathParamKey]: {
            string: event.pathParameters[pathParamKey]
          }
        }
      }, {}),
    }
  };
  const queryStringObj = !hasQueryString ? {} : {
    queryStringParameters: {
      // transform regular map into smithy format
      record: Object.keys(event.queryStringParameters).reduce((acc, queryParamKey) => {
        return {
          ...acc,
          [queryParamKey]: {
            string: event.queryStringParameters[queryParamKey]
          }
        }
      }, {}),
    }
  };
  return {
    contextMap: {
      ...queryStringObj,
      ...pathParametersObj,
    }
  };
}

async function handler(event, context) {
  // https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-known-issues.html
  // > Header names and query parameters are processed in a case-sensitive way.
  // https://www.rfc-editor.org/rfc/rfc7540#section-8.1.2
  // > header field names MUST be converted to lowercase prior to their encoding in HTTP/2
  // curl defaults to HTTP/2
  let bearerToken =
    event.headers?.Authorization || event.headers?.authorization;
  if (bearerToken?.toLowerCase().startsWith('bearer ')) {
    // per https://www.rfc-editor.org/rfc/rfc6750#section-2.1 "Authorization" header should contain:
    //  "Bearer" 1*SP b64token
    // however, match behavior of COGNITO_USER_POOLS authorizer allowing "Bearer" to be optional
    bearerToken = bearerToken.split(' ')[1];
  }
  try {
    const parsedToken = JSON.parse(Buffer.from(bearerToken.split('.')[1], 'base64').toString());
    const actionId = `${event.requestContext.httpMethod.toLowerCase()} ${event.requestContext.resourcePath}`;

    const input = {
      [tokenType]: bearerToken,
      policyStoreId: policyStoreId,
      action: {
        actionType: actionType,
        actionId: actionId,
      },
      resource: {
        entityType: resourceType,
        entityId: resourceId
      },
      context: getContextMap(event),
    };

    const authResponse = await verifiedpermissions.isAuthorizedWithToken(input);
    console.log('Decision from AVP:', authResponse.decision);
    let principalId = `${parsedToken.iss.split('/')[3]}|${parsedToken.sub}`;
    if (authResponse.principal) {
      const principalEidObj = authResponse.principal;
      principalId = `${principalEidObj.entityType}::"${principalEidObj.entityId}"`;
    }

    return {
      principalId,
      policyDocument: {
        Version: '2012-10-17',
        Statement: [
          {
            Action: 'execute-api:Invoke',
            Effect: authResponse.decision.toUpperCase() === 'ALLOW' ? 'Allow' : 'Deny',
            Resource: event.methodArn
          }
        ]
      },
      context: {
        actionId,
      }
    }
  } catch (e) {
    console.log('Error: ', e);
    return {
      principalId: '',
      policyDocument: {
        Version: '2012-10-17',
        Statement: [
          {
            Action: 'execute-api:Invoke',
            Effect: 'Deny',
            Resource: event.methodArn
          }
        ]
      },
      context: {}
    }
  }
}

module.exports = {
  handler,
};
```

The Lambda authorizer function uses the `@aws-sdk/client-verifiedpermissions` SDK to interact with the Amazon Verified Permissions service. The code defines several constants from environment variables, including `policyStoreId`, `namespace`, and `tokenType`. These are used to configure the VerifiedPermissions client and to construct the input for the `isAuthorizedWithToken` method later on.

The `getContextMap` function is a helper function that transforms the path parameters and query string parameters from the event object into a specific format (the "smithy" format). This function is used to provide context for the authorization request.

The `handler` function is the main function that handles incoming events. It first retrieves the `Authorization` header from the event, which should contain a bearer token. This token is then parsed and used as part of the input for the `isAuthorizedWithToken` method.

The `isAuthorizedWithToken` method is called with an input object that includes the bearer token, the policy store ID, the action (which includes the action type and action ID), the resource (which includes the entity type and entity ID), and the context map. This method returns a promise that resolves to an authorization response.

The authorization response includes a decision (either "ALLOW" or "DENY") and possibly a principal. The decision is used to construct a policy document that is returned by the `handler` function. The principal, if present, is used to construct a principal ID that is also included in the return object.

If an error occurs during the execution of the `handler` function, it logs the error and returns a policy document that denies access.

Summary

In this section, you reviewed the Lambda authorizer function that was automatically created in the previous section. The Lambda authorizer is responsible for authorizing users before they can access the API. The Lambda authorizer function uses the @aws-sdk/client-verifiedpermissions SDK to interact with the Amazon Verified Permissions service and validate incoming requests. You reviewed the code of the Lambda authorizer function and learned how it constructs an

Testing permissions

In this section, you will test the permissions you configured in the previous section. You will use the `alice` and `bob` users to test the permissions.

Make sure that you have up-to-date tokens in the `RE_ACCESS_TOKEN_ALICE`, `RE_ACCESS_TOKEN_BOB`, and `RE_ACCESS_TOKEN_CHARLIE` environment variables. The access and ID tokens expiration time for this wokshop is configured to 24 hours. However, since we added the users to the groups, we need to refresh the tokens, so they contain the group information. See the **Authenticating with Amazon Cognito** section for details. **Hint**: feel free to use the `ws-env.sh` script to set the environment variables and then run `. ws-env.sh` to export them to the current Cloudshell terminal.

You can check the content of "access_token" with the following command:

```
jq -R 'split(".") | .[1] | @base64d | fromjson' <<< "$RE_ACCESS_TOKEN_ALICE"
```

The output should look something like this:


```
{
  "sub": "7498a4e8-b061-7002-5d45-6742438aa8bd",
  "cognito:groups": [
    "Admin"
  ],
  "iss": "https://cognito-idp.us-east-1.amazonaws.com/us-east-1_u5FeKzadj",
  "version": 2,
  "client_id": "7ts0de0r8ara8prSmcsvhv8nlh",
  "origin_jti": "7ed1edba-6ba5-43ad-a963-c58a7f69e2ae",
  "event_id": "fe8d2b66-e31d-4b9a-b2a9-1cd2e1a931a2",
  "token_use": "access",
  "scope": "openid profile RealEstateResourceServer/RealEstateApi email",
  "auth_time": 1726354127,
  "exp": 1726440527,
  "iat": 1726354127,
  "jti": "3e5d3ee3-f6bd-436f-840e-d4c19cfa05dc",
  "username": "alice"
}
```


Notice that the token now includes the `cognito:groups` section, with the `Admin` group assigned.

Testing `alice` permissions

`alice` is a member of the `Admin` group, which has permissions to perform all operations on the RealEstate API. Let's test that.

Getting the list of pets as `alice`

To get a list of real estate properties, use the following API call:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_ALICE" | jq
```

The response looks like this:


```
[
  {
    "location": "Malibu, California",
    "description": "A beautiful beachfront villa with a stunning ocean view.",
    "id": "1bbe07a8-bab9-47e1-a7ac-668f3dd394f8",
    "price": 950000,
    "name": "Beachfront Villa"
  },
  {
    "location": "Aspen, Colorado",
    "description": "A cozy cabin in the mountains, perfect for a winter getaway.",
    "id": "2cde08a9-cdc9-48a8-bb8a-74b9e9a748ac",
    "price": 350000,
    "name": "Mountain Cabin"
  }
]
```


Adding a new real estate property

To add a new real estate property, use the following API call:

```
curl -X POST "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_ALICE" \
  -H "Content-Type: application/json" \
  -d '{
      "name": "Mountain Retreat",
      "description": "A peaceful retreat in the mountains, surrounded by nature.",
      "price": 1750000,
      "location": "Aspen, Colorado",
      "bedrooms": 5,
      "bathrooms": 3,
      "squarefeet": 3200
    }' | jq
```

The response looks like this:


```
{
  "message": "Real estate property created successfully",
  "id": "3f2b10cd-26fa-4ab8-9230-cd01cf6d9db4"
}
```


As you can see, `alice` was able to add a new real estate property.

Testing `bob` permissions

To get a list of real estate properties, use the following API call:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_BOB" | jq
```

The response should look like this:


```
[
  {
    "location": "Aspen, Colorado",
    "bedrooms": 5,
    "bathrooms": 3,
    "squarefeet": 3200,
    "price": 1750000,
    "description": "A peaceful retreat in the mountains, surrounded by nature.",
    "id": "3f2b10cd-26fa-4ab8-9230-cd01cf6d9db4",
    "name": "Mountain Retreat"
  },
  {
    "location": "Malibu, California",
    "description": "A beautiful beachfront villa with a stunning ocean view.",
    "id": "1bbe07a8-bab9-47e1-a7ac-668f3dd394f8",
    "price": 950000,
    "name": "Beachfront Villa"
  },
  {
    "location": "Aspen, Colorado",
    "description": "A cozy cabin in the mountains, perfect for a winter getaway.",
    "id": "2cde08a9-cdc9-48a8-bb8a-74b9e9a748ac",
    "price": 350000,
    "name": "Mountain Cabin"
  }
]
```


As you can see, `bob` was able to read the list of real estate properties. He was also able to see the property added by `alice`. Let's try to add a new property.

Getting a real estate property by ID as `bob`

To get a real estate property by ID, use the following API call:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties/3f2b10cd-26fa-4ab8-9230-cd01cf6d9db4" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_BOB" | jq
```

The response should look like this:


```
{
  "Message": "User is not authorized to access this resource with an explicit deny"
}
```


If you can successfully see the real estate property's details, that means that API Gateway changes are not propagated or the default authorization cache (120 seconds) that we disabled in the previous section has not invalidated yet. Wait a minute and try again. Another reason could be that you have not attached the Lambda authorizer to the GET /real-estate-properties/{id} method. Check the previous section for details.

Feel free to experiment with other API calls to test the permissions further.

Testing `charlie` permissions

`charlie` is not a member of any group. Let's see what happens when he tries to access the API.

To get a list of real estate properties, use the following API call:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_CHARLIE" | jq
```

The response should look like this:


```
{
  "Message": "User is not authorized to access this resource with an explicit deny"
}
```


If you can successfully see the real estate property's details, that means that API Gateway changes are not propagated or the default authorization cache (120 seconds) that we disabled in the previous section has not invalidated yet. Wait a minute and try again. Another reason could be that you have not attached the Lambda authorizer to the GET /real-estate-properties/{id} method. Check the previous section for details.

As you can see, `charlie` was not able to access the API. The API returned an error message saying that the user is not authorized to access this resource with an explicit deny. This is because `charlie` is not a member of any group.

Summary

In this section, you tested the permissions you configured in the previous section. You used the `alice` and `bob` users to test the permissions. You saw that `alice` was able to perform all operations on the RealEstate API, `bob` was able to read the list of real estate properties only, and `charlie` was not able to access the API.

Additional Tasks

Once you have completed the lab, you can try the following additional challenge.

Task 1: Add new permisison to `User` group

Modify the permissions of the `User` group so that users in this group are able to perform a `GET` request on `/real-estate-properties/{id}` for any property.

1. Update the Cedar policy to grant this access.
2. Test the API with `bob`, who is part of the `User` group.

List all real estate properties, and get an id:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_BOB" | jq
```

Now, try to get a real estate property by ID, use the following API call:

```
curl -X GET "$RE_API_ENDPOINT/real-estate-properties/3f2b10cd-26fa-4ab8-9230-cd01cf6d9db4" \
  -H "Authorization: Bearer $RE_ACCESS_TOKEN_BOB" | jq
```

The response should look like this:

```
{
  "location": "Aspen, Colorado",
  "bedrooms": 5,
  "bathrooms": 3,
  "squareFeet": 3200,
  "price": 1750000,
  "description": "A peaceful retreat in the mountains, surrounded by nature.",
  "id": "3f2b10cd-26fa-4ab8-9230-cd01cf6d9db4",
  "name": "Mountain Retreat"
}
```

Hint
Hint 1

▶ Expand

Hint
Hint 2

▶ Expand

Task 2: Add a new role

Add a new Cognito user group `Broker`, assign `charlie` to the group, and create a Cedar policy in Amazon Verified Permissions to allow the `Broker` group to access the following API Gateway resources:

- `GET /real-estate-properties`
- `GET /real-estate-properties/{id}`
- `POST /real-estate-properties`
- `PUT /real-estate-properties/{id}`

Test the API with `charlie` to ensure that users in the `Broker` group have the correct permissions.

Hint
Hint 1

▶ Expand

Hint
Hint 2

▶ Expand

Congratulations! You have successfully completed this module.

Previous Next